Сергей Фельдман

Системное программирование на персональном компьютере

издание второе, дополненное и исправленное

Рекомендуется в качестве дополнительного учебного пособия студентам высших учебных заведений по дисциплине «Системное программирование» и по специальности 351400 «Прикладная информатика»

УДК 004.42
ББК 32.973.26-018.2
Ф39

Рецензент:
кандидат технических наук *В. Л. Райков*
доктор технических наук *С. С. Валенчик*

Фельдман С. К.
Ф39 Системное программирование на персональном компьютере. / С. К. Фельдман. – 2-е изд. – М.: Бук-пресс, 2006.— 512 с.
ISBN 5-9643-0027-8

В этом курсе лекций излагаются классические модели, методы и алгоритмы языка программирования высокого уровня, дается строгое изложение основ теории системного программирования, приводятся примеры программ на языке Ассемблер. Главное внимание уделяется объяснению того, как использовать на практике полученные знания. Курс разбит на лекции, включающие теоретический материал и лабораторные работы.

Данное издание рекомендуется в качестве дополнительного учебного пособия студентам высших учебных заведений по дисциплине «Системное программирование» и по специальности 351400 «Прикладная информатика».

УДК 004.42
ББК 32.973.26-018.2

ISBN 5-9643-0027-8

Содержание

Лекция 1. Основные понятия и определения

Программы и программное обеспечение ...3
Системное программирование ...5
Этапы подготовки программы ...7

Лекция 2. Ассемблеры

Программирование на языке Ассемблера ...11
Предложения языка Ассемблера ...12
Регистры ...15
Биты и байты ...15
ASCII ...16
Двоичные числа ...17
Шестнадцатеричное представление ...20
Сегменты ...21
Расширение набора команд ...23
Способы адресации ...27
Директивы ...29
Директивы определения данных ...30
Директива определения байта (DB) ...33
Директива определения слова (DW) ...33
Директива определения двойного слова (DD) ...34
Директива определения учетверенного слова (DQ) ...34
Директива определения десяти байт (DT) ...35
Непосредственные операнды ...35
Директива EQU ...37

Лекция 3. Регистры

Сегментные регистры: CS, DS, SS и ES ...38
Регистры общего назначения: AX, BX, CX и DX ...39
Регистровые указатели: SP и BP ...40
Индексные регистры: SI и DI ...40
Регистр командного указателя: IP ...41
Флаговый регистр ...41

Лекция 4. Арифметические операции

Обработка двоичных данных ...43
Беззнаковые и знаковые данные ...44
Умножение ...46
Сдвиг регистровой пары DX:AX ...49
Деление ...50
Преобразование знака ...53
Обработка данных в форматах ASCII и BCD ...55
Двоично-десятичный формат (BCD) ...57
Сдвиг и округление ...59

Лекция 5. Команды обработки строк

Свойства операций над строками ...61
REP: Префикс повторения цепочечной команды ...62
MOVS: Пересылка строк ...63
LODS: Загрузка строки ...64
STOS: Запись строки ...64
CMPS: Сравнение строк ...65
SCAS: Сканирование строк ...66
Сканирование и замена ...66
Альтернативное кодирование ...67
Дублирование образца ...67

Лекция 6. Обработка таблиц

Определение таблиц .70
Прямой табличный доступ .71
Табличный поиск .72
Транслирующая команда XLAT .73
Операторы типа, длина и размеры .74

Лекция 7. Свойства операторов работы с экраном

Команда прерывания INT .76
Установка курсора .77
Очистка экрана .78
Использование символов возврата каретки,
конца строки и табуляции для вывода на экран .79
Расширенные возможности экранных операций .80
Расширенный ASCII код .85
Другие операции ввода/вывода .86
Ввод с клавиатуры по команде BIOS INT 16H .87
Функциональные клавиши .89
Цвет и графика .91

Лекция 8. Требования языка

Комментарии в программах на Ассемблере .93
Формат кодирования .93
Директивы .95
Память и регистры .99
Инициализация программы .99

Лекция 9. Ввод и выполнение программ

Ввод программы .102
Подготовка программы для выполнения .103

Ассемблирование программы .104
Компоновка программы .106
Выполнение программы .107
Файл перекрестных ссылок .108

Лекция 10. Алгоритмы работы Ассемблеров

Двухпроходный Ассемблер — первый проход .110
Структура таблиц Ассемблера .117
Двухпроходный Ассемблер — второй проход .118
Некоторые дополнительные директивы .125
Директивы связывания .126
Одно- и многопроходный Ассемблер .127

Лекция 11. Логика и организация программы

Команда JMP .129
Команда LOOP .130
Флаговый регистр .131
Команды условного перехода .133
Процедуры и оператор CALL .134
Сегмент стека .136
Команды логических операций: AND, OR, XOR, TEST, NOT .137
Изменение строчных букв на заглавные .139
Команды сдвига и циклического сдвига .139
Организация программ .142

Лекция 12. Компоновка программ

Межсегментные вызовы .145
Атрибуты EXTRN и PUBLIC .147
Компоновка программ на языке C и Ассемблере .148
Выполнение COM-программы .149

Выполнение EXE-программы 149
Функции загрузки и выполнения программы 152

Лекция 13. Выполнение программ

Начинаем работать 155
Определение данных 159
Машинная адресация 160
Определение размера памяти 162
Специальные средства отладчика 163

Лекция 14. Макросредства

Простое макроопределение 166
Использование параметров в макрокомандах 167
Комментарии 168
Использование макрокоманд в макроопределениях 169
Директива LOCAL 170
Использование библиотек макроопределений 170
Конкатенация (&) 172
Директивы повторения: REPT, IRP, IRPC 172
Условные директивы 174
Директива выхода из макроса EXITM 175
Макрокоманды, использующие IF и IFNDEF 176
Макрос, использующий IFIDN-условие 177

Лекция 15. Макропроцессоры

Основные понятия 179
Сравнение макросредств и подпрограмм 182
Некоторые возможности Макроязыка 183
Локальные переменные макроопределения 185
Присваивание значений переменным макроопределения 186
Глобальные переменные макроопределения 187
Уникальные метки 187

Операторы повторений 189
Выдача сообщения 190
Завершение обработки 191
Комментарии макроопределения 191
Макрорасширения в листинге 192
Алгоритм работы Макропроцессора 193
Библиотеки макроопределений 204
Вложенные макровызовы. Вложенные макроопределения 205
Качественное расширение возможностей 207
Структурный Ассемблер 208
Объектно-ориентированный Ассемблер 208
Переносимый машинный язык 209

Лекция 16. Загрузчики и редакторы связей

Основные понятия 210
Формат объектного модуля 214
Алгоритм работы Непосредственно Связывающего Загрузчика 216

Лекция 17. Кросс-системы

Вычислительные системы 223
Модель регистров 226
Модель оперативной памяти 227
Модель процессора 229

Лекция 18. Ошибки программирования

Классификация ошибок программирования 240
Ошибки при задании необходимых начальных условий для отдельных программ 242
Распознавание ошибок Ассемблером 243
Распространенные ошибки в драйверах ввода-вывода 244
Распространенные ошибки в программах прерывания 245

Лекция 19. Введение в макроассемблер

Состав пакета .247
Общие сведения .248
Запуск макроассемблера .249
Опции MASM .251
LINK: линкер модулей .254
SYMDEB: символьный отладчик программ .265
CREF: утилита перекрестных ссылок .295
LIB: утилита обслуживания библиотек .296
MAKE: утилита сопровождения программ .302
Сегментация программы .306
Условные директивы .314
Директивы условной генерации ошибок .317
Макросредства .319
Макродирективы .320
Блоки повторений .324
Макрооператоры .327
Директивы определения памяти .329
Скалярные данные .329
Записи .332
Структуры .334
Описание символических имен .336
Директивы управления файлами .339
Управление листингом .341
Другие директивы .344
Глобальные объявления .345
Инструкции процессоров .346
Инструкции пересылки данных .348
Инструкции общего назначения .349
Ввод/вывод .350
Адресные операции .351
Операции с флагами .352

Арифметические инструкции .353
Арифметические операции и флаги .354
Сложение .356
Вычитание .357
Умножение .359
Деление .360
Инструкции обработки бит .361
Сдвиги .363
Вращения .364
Инструкции обработки строк .365
Пересылка строк .368
Сравнение строк .369
Сканирование .370

Практикум

Двупросмотровый алгоритм .371
Однопросмотровый алгоритм .375
Реализация внутри Ассемблера .377

Лабораторные работы

Общие указания к выполнению .379
Язык C как инструмент системного программирования .379
Порядок выполнения работ .381
Содержание отчета .382
Лабораторная работа №1. Работа с символьными строками .382
Пример решения задачи .386
Лабораторная работа №2. Представление в памяти массивов и матриц .391
Лабораторная работа №4. Проверка оборудования .403
Лабораторня работа №5. Управление клавиатурой .413
Лабораторная работа №6. Управление таймером .419
Лабораторная работа №7. Управление видеоадаптером .425
Лабораторная работа №8. Главная загрузочная запись .432

Лабораторная работа №9. Дисковые структуры данных DOS 437
Лабораторная работа N10. Управление программами 452

Что нужно знать для экзамена

Функции прерывания DOS INT 21H 461
Порты 466

Что нужно знать для семинара

Справочник по директивам языка Ассемблера 468
Справочник по командам языка Ассемблера 477
Список использованной литературы 502

Лекция 1. Основные понятия и определения



Программы и программное обеспечение

Программа — это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма.

Обратить внимание: программа — это данные.

Один из основных принципов машины фон Неймана — то, что и программы, и данные хранятся в одной и той же памяти. Сохраняемая в памяти программа представляет собой некоторые коды, которые могут рассматриваться как данные. Возможно, с точки зрения программиста программа — активный компонент, она выполняет некоторые действия. Но с точки зрения процессора команды программы — это данные, которые процессор читает и интерпретирует. С другой стороны программа — это данные с точки зрения обслуживающих программ, например, с точки зрения компилятора, который на входе получает одни данные — программу на языке высокого уровня (ЯВУ), а на выходе выдает другие данные — программу в машинных кодах.

Программное обеспечение (ПО) — совокупность программ СОИ и программных документов, необходимых для их эксплуатации

Существенно, что ПО — это программы, предназначенные для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств ПО.

Необходимость документирования

По определению программы становятся ПО только при наличии документации. Конечный пользователь не может работать, не имея документации. Документация делает возможным тиражирование ПО и продажу его без его разработчика. По Бруксу ошибкой в ПО является ситуация, когда программное изделие функционирует не в соответствии со своим описанием, следовательно, ошибка в документации также является ошибкой в программном изделии.

Эффективность

ПО, рассчитанное на многократное использование (например, ОС, текстовый редактор) пишется и отлаживается один раз, а выполняется многократно. Таким образом, выгодно переносить затраты на этап производства ПО и освобождать от затрат этап выполнения, чтобы избежать тиражирования затрат.

Надежность

В том числе:

- Тестирование программы при всех допустимых спецификациях входных данных
- Защита от неправильных действий пользователя
- Защита от взлома — пользователи должны иметь возможность взаимодействия с ПО только через легальные интерфейсы.

Появление ошибок любого уровня не должно приводить к краху системы. Ошибки должны вылавливаться диагностироваться и (если их невозможно исправить) превращаться в корректные отказы.

Системные структуры данных должны сохраняться безусловно.

Сохранение целостности пользовательских данных желательно.

Возможность сопровождения

Возможные цели сопровождения — адаптация ПО к конкретным условиям применения, устранение ошибок, модификация.

Во всех случаях требуется тщательное структурирование ПО и носителем информации о структуре ПО должна быть программная документация.

Адаптация во многих случаях может быть передоверена пользователю — при тщательной отработке и описании сценариев инсталляции и настройки.

Исправление ошибок требует развитой сервисной службы, собирающей информацию об ошибках и формирующей исправляющие пакеты.

Модификация предполагает изменение спецификаций на ПО. При этом, как правило, должны поддерживаться и старые спецификации. Эволюционное развитие ПО экономит вложения пользователей.



Системное программирование

Системная программа — программа, предназначенная для поддержания работоспособности СОИ или повышения эффективности ее использования.

Прикладная программа — программа, предназначенная для решения задачи или класса задач в определенной области применения СОИ.

В соответствии с терминологией, системное программирование — это процесс разработки системных программ (в том числе, управляющих и обслуживающих).

С другой стороны, система — единое целое, состоящее из множества компонентов и множества связей между ними. Тогда системное программирование — это разработка программ сложной структуры.

Эти два определения не противоречат друг другу, так как разработка программ сложной структуры ведется именно для обеспечения работоспособности или повышения эффективности СОИ.

Подразделение ПО на системное и прикладное является до некоторой степени устаревшим. Сегодняшнее деление предусматривает по меньшей мере три градации ПО:

- Системное
- Промежуточное
- Прикладное

Промежуточное ПО (middleware) мы определяем как совокупность программ, осуществляющих управление вторичными (конструируемыми самим ПО) ресурсами, ориентированными на решение определенного (широкого) класса задач. К такому ПО относятся менеджеры транзакций, серверы БД, серверы коммуникаций и другие программные серверы. С точки зрения инструментальных средств разработки промежуточное ПО ближе к прикладному, так как не работает на прямую с первичными ресурсами, а использует для этого сервисы, предоставляемые системным ПО.

С точки зрения алгоритмов и технологий разработки промежуточное ПО ближе к системному, так как всегда является сложным программным изделием многократного и многоцелевого использования и в нем применяются те же или сходные алгоритмы, что и в системном ПО.

Современные тенденции развития ПО состоит в снижении объема как системного, так и прикладного программирования. Основная часть работы программистов выполняется в промежуточном ПО. Снижение объема системного программирования определено современными концепциями ОС, объектно-ориентированной архитектурой и архитектурой микроядра, в соответствии с которыми большая часть функций системы выносится в утилиты, которые можно отнести и к промежуточному ПО. Снижение объема прикладного программирования обусловлено тем, что современные продукты промежуточного ПО предлагают все больший набор инструментальных средств и шаблонов для решения задач своего класса.

Значительная часть системного и практически все прикладное ПО пишется на языках высокого уровня, что обеспечивает сокращение расходов на их разработку/модификацию и переносимость.

Системное ПО подразделяется на системные управляющие программы и системные обслуживающие программы.

Управляющая программа — системная программа, реализующая набор функций управления, который включает в себя управление ресурсами и взаимодействие с внешней средой СОИ, восстановление работы системы после проявления неисправностей в технических средствах.

Программа обслуживания (утилита) — программа, предназначенная для оказания услуг общего характера пользователям и обслуживающему персоналу СОИ.

Управляющая программа совместно с набором необходимых для эксплуатации системы утилит составляют **операционную систему** (ОС).

Кроме входящих в состав ОС утилит могут существовать и другие утилиты (того же или стороннего производителя), выполняющие дополнительное (опционное) обслуживание. Как правило, это утилиты, обеспечивающие разработку программного обеспечения для операционной системы.

Система программирования — система, образуемая языком программирования, компилятором или интерпретатором программ, представленных на этом языке, соответствующей документацией, а также вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.



Этапы подготовки программы

При разработке программ, а тем более — сложных, используется принцип модульности, разбиения сложной программы на составные части, каждая из которых может подготавливаться отдельно. Модульность является основным инструментом структурирования программного изделия, облегчающим его разработку, отладку и сопровождение.

Программный модуль — программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в оперативную память.

При выборе модульной структуры должны учитываться следующие основные соображения:

- **Функциональность** — модуль должен выполнять законченную функцию
- **Несвязность** — модуль должен иметь минимум связей с другими модулями, связь через глобальные переменные и области памяти нежелательна
- **Специфицируемость** — входные и выходные параметры модуля должны четко формулироваться

Программа пишется в виде исходного модуля.

Исходный модуль — программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целое, достаточное для проведения трансляции.

Первым (не для всех языков программирования обязательным) этапом подготовки программы является обработка ее Макропроцессором (или Препроцессором). Макропроцессор обрабатывает текст программы и на выходе его получается новая редакция текста. В большинстве систем программирования Макропроцессор совмещен с транслятором, и для программиста его работа и промежуточный ИМ «не видны».

Следует иметь в виду, что Макропроцессор выполняет обработку текста, это означает, с одной стороны, что он «не понимает» операторов языка программирования и «не знает» переменных программы, с другой, что все операторы и переменные Макроязыка (тех выражений в программе, которые адресованы Макропроцессору) в промежуточном ИМ уже отсутствуют и для дальнейших этапов обработки «не видны».

Так, если Макропроцессор заменил в программе некоторый текст A на текст B, то транслятор уже видит только текст B, и не знает, был этот текст написан программистом «своей рукой» или подставлен Макропроцессором.

Следующим этапом является трансляция.

Трансляция — преобразование программы, представленной на одном языке программирования, в программу на другом языке программирования, в определенном смысле равносильную первой.

Как правило, выходным языком транслятора является машинный язык целевой вычислительной системы. (Целевая ВС — та ВС, на которой программа будет выполняться.)

Машинный язык — язык программирования, предназначенный для представления программы в форме, позволяющей выполнять ее непосредственно техническими средствами обработки информации.

Трансляторы — общее название для программ, осуществляющих трансляцию. Они подразделяются на Ассемблеры и Компиляторы — в зависимости от исходного языка программы, которую они обрабатывают. Ассемблеры работают с Автокодами или языками Ассемблера, Компиляторы — с языками высокого уровня.

Автокод — символьный язык программирования, предложения которого по своей структуре в основном подобны командам и обрабатываемым данным конкретного машинного языка.

Язык Ассемблера — язык программирования, который представляет собой символьную форму машинного языка с рядом возможностей, характерных для языка высокого уровня (обычно включает в себя макросредства).

Язык высокого уровня — язык программирования, понятия и структура которого удобны для восприятия человеком.

Объектный модуль — программный модуль, получаемый в результате трансляции исходного модуля.

Поскольку результатом трансляции является модуль на языке, близком к машинному, в нем уже не остается признаков того, на каком исходном языке был написан программный модуль. Это создает принципиальную возможность создавать программы из модулей, написанных на разных языках. Специфика исходного языка, однако, может сказываться на физическом представлении базовых типов данных, способах обращения к процедурам/функциям и т.п. Для совместимости разноязыковых модулей должны выдерживаться общие соглашения. Большая часть объектного модуля — команды и данные машинного языка именно в той форме, в какой они будут существовать во время выполнения программы. Однако, программа в общем случае состоит из многих модулей. Поскольку транслятор обрабатывает только один конкретный модуль, он не может должным образом обработать те части этого модуля, в которых запрограммированы обращения к данным или процедурам, определенным в другом модуле. Такие обращения называются внешними ссылками. Те места в объектном модуле, где содержатся внешние ссылки, транслируются в некоторую промежуточную форму, подлежащую дальнейшей обработке. Говорят, что объектный модуль представляет собой программу на машинном языке с неразрешенными внешними ссылками. Разрешение внешних ссылок выполняется на следующем этапе подготовки, который обеспечивается Редактором Связей (Компоновщиком). Редактор Связей соединяет вместе все объектные модули, входящие в программу. Поскольку Редактор Связей «видит» уже все компоненты программы, он имеет возможность обработать те места в объектных модулях, которые содержат внешние ссылки. Результатом работы Редактора Связей является загрузочный модуль.

Загрузочный модуль — программный модуль, представленный в форме, пригодной для загрузки в оперативную память для выполнения.

Загрузочный модуль сохраняется в виде файла на внешней памяти. Для выполнения программа должна быть перенесена (загружена) в оперативную память. Иногда при этом требуется некоторая дополнительная обработка (например, настройка адресов в программе на ту область оперативной памяти, в которую программа загрузилась). Эта функция выполняется Загрузчиком, который обычно входит в состав операционной системы. Возможен также вариант, в котором редактирование связей выполняется при каждом запуске программы на выполнение и совмещается с загрузкой. Это делает Связывающий Загрузчик. Вариант связывания при запуске более расходный, т.к. затраты на связывание тиражируются при каждом запуске. Но он обеспечивает:

- большую гибкость в сопровождении, так как позволяет менять отдельные объектные модули программы, не меняя остальных модулей;
- экономию внешней памяти, т.к. объектные модули, используемые во многих программах не копируются в каждый загрузочный модуль, а хранятся в одном экземпляре.

Вариант интерпретации подразумевает прямое исполнение исходного модуля.

Интерпретация — реализация смысла некоторого синтаксически законченного текста, представленного на конкретном языке.

Интерпретатор читает из исходного модуля очередное предложение программы, переводит его в машинный язык и выполняет. Все затраты на подготовку тиражируются при каждом выполнении, следовательно, интерпретируемая программа принципиально менее эффективна, чем транслируемая. Однако, интерпретация обеспечивает удобство разработки, гибкость в сопровождении и переносимость.

Не обязательно подготовка программы должна вестись на той же вычислительной системе и в той же операционной среде, в которых программа будет выполняться. Системы, обеспечивающие подготовку программ в среде, отличной от целевой называются кросс-системами. В кросс-системе может выполняться вся подготовка или ее отдельные этапы:

- Макрообработка и трансляция
- Редактирование связей
- Отладка

Типовое применение кросс-систем — для тех случаев, когда целевая вычислительная среда просто не имеет ресурсов, необходимых для подготовки программ, например, встроенные системы. Программные средства, обеспечивающие отладку программы на целевой системе можно также рассматривать как частный случай кросс-системы.

Лекция 2. Ассемблеры



Программирование на языке Ассемблера

Язык Ассемблера — система записи программы с детализацией до отдельной машинной команды, позволяющая использовать мнемоническое обозначение команд и символическое задание адресов.

Поскольку в разных аппаратных архитектурах разные программно-доступные компоненты (система команд, регистры, способы адресации), язык Ассемблера аппаратно-зависимый. Программы, написанные на языке Ассемблера могут быть перенесены только на вычислительную систему той же архитектуры.

Программирование на языке Ассемблера позволяет в максимальной степени использовать особенности архитектуры вычислительной системы. До недавнего времени воспринималась как аксиома, что ассемблерная программа всегда является более эффективной и в смысле быстродействия, и в смысле требований к памяти. Для Intel-архитектуры это и сейчас так.

Но это уже не так для RISK-архитектур. Для того, чтобы программа могла эффективно выполняться в вычислительной среде с распараллеливанием на уровне команд, она должна быть определенным образом оптимизирована, то есть, команды должны быть расположены в определенном порядке, допускающим их параллельное выполнение. Программист просто не сможет покомандно оптимизировать всю свою программу. С задачей такой оптимизации более эффективно справляются компиляторы.

Доля программ, которые пишутся на языках Ассемблеров в мире, неуклонно уменьшается, прикладное программирование на языках Ассемблеров применяется только по недомыслию. Язык Ассемблера «в чистом виде» применяется только для написания отдельных небольших частей системного ПО: микроядра ОС, самых нижних уровней драйверов — тех частей, которые непосредственно взаимодействуют с реальными аппаратными компонентами.

Этим занимается узкий круг программистов, работающих в фирмах, производящих аппаратуру и ОС. Зачем же нам тогда изучать построение Ассемблера?

Хотя разработка программ, взаимодействующих с реальными аппаратными компонентами, — редкая задача, в современном программировании при разработке прикладного, а еще более — промежуточного ПО довольно часто применяется технологии виртуальных машин. Для выполнения того или иного класса задач программно моделируется некоторое виртуальное вычислительное устройство, функции которого соответствуют нуждам этого класса задач.

Для управления таким устройством для него может быть создан соответствующий язык команд. (Широко известные примеры: MI AS/400, JVM.) Говоря шире, любую программу можно представить себе как виртуальное «железо», решающее конкретную задачу. (Конечный пользователь обычно не видит разницы между программой и аппаратурой и часто говорит не «мне программа выдала то-то», а «мне компьютер выдал то-то»). В некоторых случаях интерфейс программы удобно представить в виде системы команд, а следовательно, нужен соответствующий Ассемблер. (Это, конечно, относится не к программам «для чайников», а к инструментальным средствам программистов, системам моделирования).



Предложения языка Ассемблера

Предложения языка Ассемблера описывают команды или псевдокоманды (директивы). Предложения-команды задают машинные команды вычислительной системы; обработка Ассемблером команды приводит к генерации машинного кода. Обработка псевдокоманды не приводит к непосредственной генерации кода, псевдокоманда управляет работой самого Ассемблера. Для одной и той же аппаратной архитектуры могут быть построены разные Ассемблеры, в которых команды будут обязательно одинаковые, но псевдокоманды могут быть разные.

Во всех языках Ассемблеров каждое новое предложение языка начинается с новой строки. Каждое предложение, как правило, занимает одну строку, хотя обычно допускается продолжение на следующей строке/строках. Формат записи предложений языка м.б. жесткий или свободный. При записи в жестком формате составляющие предложения должны располагаться в фиксированных позициях строки. (Например: метка должна располагаться в позициях 1-8, позиция 9 — пустая, позиции 10-12 — мнемоника команды, позиция 13 — пустая, начиная с позиции 14 — операнды, позиция 72 — признак продолжения). Обычно для записи программ при жестком формате создаются бланки. Жесткий формат удобен для обработки Ассемблером (удобен и для чтения).

Свободный формат допускает любое количество пробелов между составляющими предложения.

В общих случаях предложения языка Ассемблера состоят из следующих компонент:

- метка или имя;
- мнемоника;
- операнды;
- комментарии.

Метка или имя является необязательным компонентом. Не во всех языках Ассемблеров эти понятия различаются. Если они различаются (например, MASM), то метка — точка программы, на которую передается управление, следовательно, метка стоит в предложении, содержащем команду; имя — имя переменной программы, ячейки памяти, следовательно, имя стоит в предложении, содержащем псевдокоманду резервирования памяти или определения константы. В некоторых случаях метка и имя могут отличаться даже синтаксически, так, в MASM/TASM после метки ставится двоеточие, а после имени — нет.

Однако, физический смысл и метки, и имени — одинаков, это — адрес памяти. Во всех случаях, когда Ассемблер встречает в программе имя или метку, он заменяет ее на адрес той ячейки памяти, к которую имя/метка именует.

Правила формирования имен/меток совпадают с таковыми для языков программирования. В некоторых Ассемблерах (HLAM S/390) не делается различия между меткой и именем.

В языке должны предусматриваться некоторые специальные правила, позволяющие Ассемблеру распознать и выделить метку/имя, например:

- метка/имя должна начинаться в 1-й позиции строки
- если метки/имени нет, то в 1-й позиции должен быть пробел, или за меткой/именем должно следовать двоеточие, и т.п.

Мнемоника — символическое обозначение команды/псевдокоманды.

Операнды — один или несколько операндов, обычно разделяемые запятыми. Операндами команд являются имена регистров, непосредственные операнды, адреса памяти (задаваемые в виде констант, литералов, символических имен или сложных выражений, включающих специальный синтаксис). Операнды псевдокоманд могут быть сложнее и разнообразнее.

Комментарии — любой текст, который игнорируется Ассемблером. Комментарии располагаются в конце предложения и отделяются от текста предложения, обрабатываемого Ассемблером, каким-либо специальным символом (в некоторых языках — пробелом). Всегда предусматривается возможность строк, содержащих только комментарий, обычно такие строки содержат специальный символ в 1-й позиции.

Константы — могут представлять непосредственные операнды или абсолютные адреса памяти. Применяются 10-е, 8-е, 16-е, 2-е, символьные константы.

Непосредственные операнды — записываются в сам код команды.

Имена — адреса ячеек памяти.

При трансляции Ассемблер преобразует имена в адреса. Способ преобразования имени в значение зависит от принятых способов адресации. Как правило, в основным способом адресации в машинных языках является адресация относительная: адрес в команде задается в виде смещения относительно какого-то базового адреса, значение которого содержится в некотором базовом регистре. В качестве базового могут применяться либо специальные регистры (DS, CS в Intel) или регистры общего назначения (S/390).

Литералы — записанные в особой форме константы. Концептуально литералы — те же имена. При появлении в программе литерала Ассемблер выделяет ячейку памяти и записывает в нее заданную в литерале константу. Далее все появления этого литерала Ассемблер заменяет на обращения по адресу этой ячейки. Таким образом, литеральные константы, хранятся в памяти в одном экземпляре, независимо от числа обращений к ним.

Специальный синтаксис — явное описание способа адресации (например, указание базового регистра и смещения).



Регистры

Программа в машинном коде состоит из различных сегментов для определения данных, для машинных команд и для сегмента, названного стеком, для хранения адресов. Для выполнения арифметических действий, пересылки данных и адресации компьютер имеет ряд регистров.

Для выполнения программ компьютер временно записывает программу и данные в основную память. Компьютер имеет также ряд регистров, которые он использует для временных вычислений.



Биты и байты

Минимальной единицей информации в компьютере является бит. Бит может быть выключен, так что его значение есть нуль, или включен, тогда его значение равно единице. Единственный бит не может представить много информации в отличие от группы битов.

Группа из девяти битов представляет собой байт; восемь битов которого содержат данные и один бит — контроль на четность. Восемь битов обеспечивают основу для двоичной арифметики и для представления символов, таких как буква А или символ *. Восемь битов дают 256 различных комбинаций включенных и выключенных состояний: от «все выключены» (00000000) до «все включены» (11111111). Например, сочетание включенных и выключенных битов для представления буквы А выглядит как 01000001, а для символа * — 00101010 (это можно не запоминать). Каждый байт в памяти компьютера имеет уникальный адрес, начиная с нуля.

Требование контроля на четность заключается в том, что количество включенных битов в байте всегда должно быть не четно. Контрольный бит для буквы **A** будет иметь значение единица, а для символа ***** — ноль. Когда команда обращается к байту в памяти, компьютер проверяет этот байт. В случае, если число включенных битов является четным, система выдает сообщение об ошибке. Ошибка четности может явится результатом сбоя оборудования или случайным явлением, в любом случае, это бывает крайне редко.

Откуда компьютер «знает», что значения бит **01000001** представляют букву **A**? Когда на клавиатуре нажата клавиша **A**, система принимает сигнал от этой конкретной клавиши в байт памяти. Этот сигнал устанавливает биты в значения **01000001**. Можно переслать этот байт в памяти и, если передать его на экран или принтер, то будет сгенерирована буква **A**.

По соглашению биты в байте пронумерованы от 0 до 7 справа налево:

Номера бит:	7 6 5 4 3 2 1 0
Значения бит:	0 1 0 0 0 0 0 0

Число 2 в десятой степени равно 1024, что составляет один килобайт и обозначается буквой **K**. Например, компьютер с памятью в 512 K содержит 512 x 1024, то есть, 524288 байт. Процессор в PC и в совместимых моделях использует 16-битовую архитектуру, поэтому он имеет доступ к 16-битовым значениям как в памяти, так и в регистрах. 16-битовое (двухбайтовое) поле называется словом. Биты в слове пронумерованы от 0 до 15 справа налево.



ASCII

Для целей стандартизации в микрокомпьютерах используется американский национальный стандартный код для обмена информацией ASCII (American National Standard Code for Information Interchange). Читается как «аски» код. Именно по этой причине комбинация бит 01000001 обозначает букву А.

Наличие стандартного кода облегчает обмен данными между различными устройствами компьютера. 8-битовый расширенный ASCII-код, используемый в PC обеспечивает представление 256 символов, включая символы для национальных алфавитов.



Двоичные числа

Так как компьютер может различить только нулевое и единичное состояние бита, то он работает системе исчисления с базой 2 или в двоичной системе. Фактически бит унаследовал свое название от английского «Binary digit» (двоичная цифра).

Сочетанием двоичных цифр (битов) можно представить любое значение. Значение двоичного числа определяется относительной позицией каждого бита и наличием единичных битов.

Самый правый бит имеет весовое значение 1, следующая цифра влево — 2, следующая — 4 и так далее. Общая сумма для восьми единичных битов в данном случае составит 1+2+4+...+128, или 255 (2 в восьмой степени — 1).

Для двоичного числа 01000001 единичные биты представляют значения 1 и 64, то есть, 65. Но 01000001 представляет также букву А! Действительно, здесь момент, который необходимо четко уяснить. Биты 01000001 могут представлять как число 65, так и букву А:

- если программа определяет элемент данных для арифметических целей, то 01000001 представляет двоичное число эквивалентное десятичному числу 65;
- если программа определяет элемент данных (один или более смежных байт), имея в виду описательный характер, как, например, заголовок, тогда 01000001 представляет собой букву или «строку».

При программировании это различие становится понятным, так как назначение каждого элемента данных определено.

Двоичное число не ограничено только восемью битами. Процессор может использовать 16-битовую архитектуру, в этом случае он автоматически оперирует с 16-битовыми числами. 2 в степени 16 минус 1 дает значение 65535, а немного творческого программирования позволит обрабатывать числа до 32 бит (2 в степени 32 минус 1 равно 4294967295) и даже больше.

Двоичная арифметика

Компьютер выполняет арифметические действия только в двоичном формате. Поэтому программист на языке Ассемблера должен быть знаком с двоичным форматом и двоичным сложением:

```
0 + 0 = 0
1 + 0 = 1
1 + 1 = 10
1 + 1 + 1 = 11
```

Отрицательные числа

Все представленные выше двоичные числа имеют положительные значения, что обозначается нулевым значением самого левого (старшего) разряда. Отрицательные двоичные числа содержат единичный бит в старшем разряде и выражаются двоичным дополнением. То есть, для представления отрицательного двоичного числа необходимо инвертировать все биты и прибавить 1.

Рассмотрим пример:

Число 65: 01000001

Инверсия: 10111110

Плюс 1: 10111111 (равно -65).

В случае, если прибавить единичные значения к числу 10111111, 65 не получится.

Фактически двоичное число считается отрицательным, если его старший бит равен 1. Для определения абсолютного значения отрицательного двоичного числа, необходимо повторить предыдущие операции: инвертировать все биты и прибавить 1:

Двоичное значение: 10111111

Инверсия: 01000000

Плюс 1: 01000001 (равно +65).

Сумма +65 и -65 должна составить ноль:

01000001 (+65) + 10111111 (-65) = (1) 00000000

Все восемь бит имеют нулевое значение. Перенос единичного бита влево потерян. Однако, если был перенос в знаковый разряд и из разрядной сетки, то результат является корректным.

Двоичное вычитание выполняется просто: инвертируется знак вычитаемого и складываются два числа. Вычтем, например, 42 из 65. Двоичное представление для 42 есть 00101010, и его двоичное дополнение: — 11010110:

65 01000001 +(-42) 11010110 = 23 (1) 00010111

Результат 23 является корректным. В рассмотренном примере произошел перенос в знаковый разряд и из разрядной сетки.

В случае, если справедливость двоичного дополнения не сразу понятна, рассмотрим следующие задачи: Какое значение необходимо прибавить к двоичному числу 00000001, чтобы получить число 00000000? В терминах десятичного исчисления ответом будет -1. Для двоичного рассмотрим 11111111:

00000001 11111111 Результат: (1) 00000000

Игнорируя перенос (1), можно видеть, что двоичное число 11111111 эквивалентно десятичному -1 и соответственно:

0 00000000 -(+1) -00000001 -1 11111111

Можно видеть также каким образом двоичными числами представлены уменьшающиеся числа:

+3 00000011

+2 00000010

+1 00000001

0 00000000

-1 11111111

-2 11111110

-3 11111101

Фактически нулевые биты в отрицательном двоичном числе определяют его величину: рассмотрите позиционные значения нулевых битов как если это были единичные биты, сложите эти значения и прибавьте единицу.



Шестнадцатеричное представление

Представим, что необходимо просмотреть содержимое некоторых байт в памяти. Требуется определить содержимое четырех последовательных байт (двух слов), которые имеют двоичные значения. Так как четыре байта включают в себя 32 бита, то специалисты разработали «стенографический» метод представления двоичных данных. По этому методу каждый байт делится пополам и каждые полбайта выражаются соответствующим значением. Рассмотрим следующие четыре байта:

Двоичное:	0101	1001	0011	0101	1011	1001	1100	1110
Десятичное:	5	9	3	5	11	9	12	14

Так как здесь для некоторых чисел требуется две цифры, расширим систему счисления так, чтобы 10=A, 11=B, 12=C, 13=D, 14=E, 15=F. Таким образом получим более сокращенную форму, которая представляет содержимое вышеуказанных байт:

```
59 35 B9 CE
```

Такая система счисления включает «цифры» от 0 до F, и так как таких цифр 16, она называется шестнадцатеричным представлением.

Шестнадцатеричный формат нашел большое применение в языке Ассемблера. В листингах ассемблирования программ в шестнадцатеричном формате показаны все адреса, машинные коды команд и содержимое констант. Также для отладки при использовании программы DOS DEBUG адреса и содержимое байтов выдается в шестнадцатеричном формате.

В случае, если немного поработать с шестнадцатеричным форматом, то можно быстро привыкнуть к нему. Рассмотрим несколько простых примеров шестнадцатеричной арифметики. Следует помнить, что после шестнадцатеричного числа F следует шестнадцатеричное 10, что равно десятичному числу 16.

Заметьте также, что шестнадцатеричное 20 эквивалентно десятичному 32, шестнадцатеричное 100 — десятичному 256 и шестнадцатеричное 100 — десятичному 4096.

Шестнадцатеричные числа записываются, например, как **шест. 4B**, двоичные числа как **дв.01001011**, а десятичные числа, как **75** (отсут-

ствие какого-либо описания предполагает десятичное число). Для индикации шестнадцатеричные числа в ассемблерной программе непосредственно после числа ставится символ **H**, например, **25H** (десятичное значение 37). Шестнадцатеричное число всегда начинается с десятичной цифры от 0 до 9, таким образом, **B8H** записывается как **0B8H**.



Сегменты

Сегментом называется область, которая начинается на границе параграфа, то есть, по любому адресу, который делится на 16 без остатка. Хотя сегмент может располагаться в любом месте памяти и иметь размер до 64 Кбайт, он требует столько памяти, сколько необходимо для выполнения программы.

Сегмент кодов

Сегмент кодов содержит машинные команды, которые будут выполняться. Обычно первая выполняемая команда находится в начале этого сегмента и операционная система передает управление по адресу данного сегмента для выполнения программы.

Регистр сегмента кодов (CS) адресует данный сегмент.

Сегмент данных

Сегмент данных содержит определенные данные, константы и рабочие области, необходимые программе. Регистр сегмента данных (DS) адресует данный сегмент.

Сегмент стека

Стек содержит адреса возврата как для программы для возврата в операционную систему, так и для вызовов подпрограмм для возврата в главную программу. Регистр сегмента стека (SS) адресует данный сегмент.

Еще один сегментный регистр, регистр дополнительного сегмента (ES), предназначен для специального использования. Последовательность регистров и сегментов на практике может быть иной.

Три сегментных регистра содержат начальные адреса соответствующих сегментов и каждый сегмент начинается на границе параграфа.

Внутри программы все адреса памяти относительны к началу сегмента. Такие адреса называются смещением от начала сегмента. Двухбайтовое смещение (16-бит) может быть в пределах от шест.0000 до шест.FFFF или от 0 до 65535. Для обращения к любому адресу в программе, компьютер складывает адрес в регистре сегмента и смещение. Например, первый байт в сегменте кодов имеет смещение 0, второй байт — 01 и так далее до смещения 65535.

В качестве примера адресации, допустим, что регистр сегмента данных содержит шест.045F и некоторая команда обращается к ячейке памяти внутри сегмента данных со смещением 0032. Несмотря на то, что регистр сегмента данных содержит 045F, он указывает на адрес 045F0, то есть, на границе параграфа. Действительный адрес памяти поэтому будет следующий:

Адрес в DS: 045F0

Смещение: 0032

Реальный адрес: 04622

Каким образом процессоры адресуют память в один миллион байт?

В регистре содержится 16 бит. Так как адрес сегмента всегда на границе параграфа, младшие четыре бита адреса равны нулю.

Шест.FFF0 позволяет адресовать до 65520 (плюс смещение) байт. Но специалисты решили, что нет смысла иметь место для битов, которые всегда равны нулю.

Поэтому адрес хранится в сегментном регистре как шест. nnnn, а компьютер полагает, что имеются еще четыре нулевых младших бита (одна шест. цифра), то есть, шест.nnnn0. Таким образом, шест.FFFF0 позволяет адресовать до 1048560 байт.

В случае, если вы сомневаетесь, то декодируйте каждое шест.F как двоичное 1111, учтите нулевые биты и сложите значения для единичных бит.



Расширение набора команд

Команды делятся на следующие группы:

- арифметические;
- логические;
- передачи данных;
- перехода;
- пропуска;
- вызова подпрограммы;
- возврата из подпрограммы;
- смешанные.

Типы операндов для каждого типа команд обсуждаются в соответствующем порядке:

- байт;
- слово;
- десятичный операнд;
- разряд;
- число;
- составной операнд.

При обсуждении способов адресации используется следующий порядок:

- прямая;
- косвенная;
- непосредственная;
- индексная;
- регистровая;
- автоиндексирование с предварительным увеличением адреса;

- автоиндексирование с предварительным уменьшением адреса;
- автоиндексирование с последующем уменьшением адреса;
- косвенная с предварительным индексированием;
- косвенная с последующем индексированием.

Арифметические команды

В эту группу включены следующие команды:

- сложение;
- сложение с флагом переноса;
- вычитание;
- вычитание при перестановке операндов;
- вычитание с флагом переноса (заем);
- увеличение на 1;
- уменьшение на 1;
- умножение;
- деление;
- сравнение;
- получение дополнения до двух (отрицательного числа);
- расширение.

Для удобства те команды, принадлежность которых к конкретной категории неясна, повторяются во всех категориях, к которым они могли бы быть отнесены.

Логические команды

Эта группа включает следующие команды:

- логическое **И**
- логическое **ИЛИ**
- логическое **исключающее ИЛИ**
- логическое **НЕ** (дополнение)
- сдвиг
- циклический сдвиг

- проверку.

Она включает также те арифметические команды (такие, как сложение с аккумулятора с самим собой), которые выполняют логические функции.

Команды передачи данных

Эта группа включает команды:

- загрузки;
- запоминания;
- пересылки;
- обмена;
- ввода;
- вывода;
- очистки;
- установки.

Кроме того, она включает арифметические команды (такие как вычитание аккумулятора из самого себя), которые заносят определенное значение или содержимое какого-либо регистра в аккумулятора или другой регистр назначения, не изменяя при этом данных.

Команды перехода

Эта группа включает следующие виды переходов:

Команды безусловного перехода

- Перейти косвенно;
- Перейти по индексу, предполагая, что базовый адрес таблицы адресов находится в регистрах **H** и **L**, а индекс в аккумуляторе;
- Перейти и связать, то есть, передать управление по адресу **DEST**, сохранив текущее состояние счетчика команд в регистрах **H** и **L**.

Команды условного перехода

- Перейти при равенстве нулю;
- Перейти при неравенстве нулю;
- Перейти, если значения равны;

- Перейти, если значения не равны;
- Перейти, если значение положительное;
- Перейти, если значение отрицательное;
- Переходы с учетом знака;
- Перейти, если больше (без учета знака), то есть, если операнды не равны и при сравнении не требуется заема;
- Перейти, если значение не больше (без учета знака), то есть, если сравниваемые операнды равны или при их сравнении требуется заем;
- Перейти, если значение меньше (без учета знака), то есть, если сравнение без знака требует заема;
- Перейти, если значение не меньше (без учета знака), то есть, если сравнение без знака не требует заема.

Команды пропуска

Команда пропуска может быть выполнена с помощью команды перехода с соответствующем адресом назначения.

Этот адрес назначения должен указывать на команду, следующую после той, которая стоит непосредственно за командой перехода.

Действительное число пропускаемых байтов будет меняться, так как команды могут иметь длину 1-3 байта.

Команды вызова подпрограмм и возврата из подпрограмм

Команда безусловного вызова

Косвенный вызов может быть выполнен с помощью обращения к промежуточной подпрограмме, которая переходит косвенно на вызываемую подпрограмму.

Команда условного вызова

Условный вызов подпрограммы может быть выполнен с помощью последовательностей команд для условного перехода.

Единственное отличие состоит в том, что команды перехода к действительным адресам назначения должны быть заменены на команды вызова подпрограмм.

Команды возврата из подпрограмм разделяются на:

- Команды безусловного возврата
- Команды условного возврата
- Команды возврата с пропуском
- Команды возврата после прерывания

Смешанные команды

В эту категорию входят следующие команды:

- нет операции
- запись в стек
- получение из стека
- останов
- ожидание
- захват (программное прерывание)
- другие, не попавшие в описание ранее категории команд.



Способы адресации

Косвенная адресация

Косвенную адресацию можно выполнить с помощью загрузки косвенных адресов в регистры **H** и **L**, используя команду **LHLD**. После этого обращение к регистру **M** является эквивалентом косвенной операции.

Таким образом, этот процесс всегда включает два шага. Кроме того, можно использовать также пары регистров **B** и **D** в командах **LDAX** и **STAX**.

Индексная адресация

Индексную адресацию можно выполнить, добавляя индекс с помощью команды **DAD** к базе. Понятно, что программное сложение требует дополнительного времени выполнения.

Предувеличение

При предувеличении адресный регистр перед использованием автоматически увеличивается.

Предувеличение может быть реализовано с помощью увеличения пары регистров перед ее использованием в качестве адреса.

Послеувеличение

При послеувеличении адресный регистр после использования в команде автоматически увеличивается.

Послеувеличение может быть реализовано с помощью увеличения пары регистров после ее использования в качестве адреса.

Предуменьшение

При предуменьшении адресный регистр перед использованием автоматически уменьшается.

Предуменьшение может быть выполнено с помощью уменьшения пары регистров перед ее использованием в качестве адреса.

Послеуменьшение

При послеуменьшении адресный регистр после использования автоматически уменьшается.

Послеуменьшение может быть выполнено с помощью уменьшения пары регистров после использования ее в качестве адреса.

Косвенная адресация с предварительным индексированием (предындексирование)

При предындексировании процессор должен сначала вычислить индексный адрес, а затем использовать этот адрес косвенно.

Так как таблица, для которой производится индексирование, должна содержать двухбайтные косвенные адреса, индексирование должно сопровождаться умножением на 2.

Косвенная адресация с последующим индексированием (послеиндексирование)

При послеиндексировании процессор должен сначала получить косвенный адрес, а затем использовать его как базу для индексирования.



Директивы

Директивы являются указаниями Ассемблеру о том, как проводить ассемблирование.

Директив может быть великое множество. В 1-м приближении мы рассмотрим лишь несколько практически обязательных директивы (мнемоники директив везде — условные, в конкретных Ассемблерах те же по смыслу директивы могут иметь другие мнемоники).

EQU
Определение имени

Перед этой директивой обязательно стоит имя. Операнд этой директивы определяет значение имени.

Операндом может быть и выражение, вычисляемое при ассемблировании. Имя может определяться и через другое имя, определенное выше. Как правило, не допускается определение имени со ссылкой вперед.

DD
Определение данных

Выделяются ячейки памяти и в них записываются значения, определяемые операндом директивы.

Перед директивой может стоять метка/имя. Как правило, одной директивой могут определяться несколько объектов данных.

В конкретных Ассемблерах может существовать либо одна общая директива DD, тогда тип данных, размещаемых в памяти определяется формой записи операндов, либо несколько подобных директив — для разных типов данных.

В отличие от других, эта директива приводит непосредственной к генерации некоторого выходного кода — значений данных.

BSS
Резервирование памяти

Выделяются ячейки памяти, но значения в них не записываются. Объем выделяемой памяти определяется операндом директивы.

Перед директивой может стоять метка/имя.

END
Конец программного модуля

Указание Ассемблеру на прекращение трансляции. Обычно в модуле, являющемся главным (main) операндом этой директивы является имя точки, на которую передается управление при начале выполнения программы. Во всех других модулях эта директива употребляется без операндов.



Директивы определения данных

Сегмент данных предназначен для определения констант, рабочих полей и областей для ввода-вывода. В соответствии с имеющимися директивами в Ассемблере разрешено определение данных различной длины: например, директива DB определяет байт, а директива DW определяет слово. Элемент данных может содержать непосредственное значение или константу, определенную как символьная строка или как числовое значение.

Другим способом определения константы является непосредственное значение, то есть, указанное прямо в ассемблерной команде, например:

```
MOV AL,20H
```

В этом случае шестнадцатеричное число 20 становится частью машинного объектного кода. Непосредственное значение ограничено одним байтом или одним словом, но там, где оно может быть применено, оно является более эффективным, чем использование константы.

Ассемблер обеспечивает два способа определения данных: во-первых, через указание длины данных и, во-вторых, по их содержимому. Рассмотрим основной формат определения данных:

```
[имя] Dn выражение
```

Имя элемента данных не обязательно (это указывается квадратными скобками), но если в программе имеются ссылки на некоторый элемент, то это делается посредством имени.

Для определения элементов данных имеются следующие директивы: DB (байт), DW (слово), DD (двойное слово), DQ (учетверенное слово) и DT (десять байт).

Выражение может содержать константу, например:

```
FLD1 DB 25
```

или знак вопроса для неопределенного значения, например

```
FLDB DB ?
```

Выражение может содержать несколько констант, разделенных запятыми и ограниченными только длиной строки:

```
FLD3 DB 11, 12, 13, 14, 15, 16, ...
```

Ассемблер определяет эти константы в виде последовательности смежных байт.

Ссылка по имени FLD3 указывает на первую константу, 11, по FLD3+1 — на вторую, 12. (FLD3 можно представить как FLD3+0). Например команда

```
MOV AL,FLD3+3
```

загружает в регистр AL значение 14 (шест. 0E). Выражение допускает также повторение константы в следующем формате:

```
[имя] Dn число-повторений DUP (выражение) ...
```

Следующие три примера иллюстрируют повторение:

```
DW 10 DUP(?) ;Десять неопределенных слов
DB 5 DUP(14) ;Пять байт, содержащих шест.14
DB 3 DUP(4 DUP(8));Двенадцать восьмерок
```

В третьем примере сначала генерируется четыре копии десятичной 8 (8888), и затем это значение повторяется три раза, давая в результате двенадцать восьмерок.

Выражение может содержать символьную строку или числовую константу.

Символьные строки

Символьная строка используются для описания данных, таких как, например, имена людей или заголовки страниц. Содержимое строки отмечается одиночными кавычками, например, 'PC' или двойными кавычками — "PC".

Ассемблер переводит символьные строки в объектный код в обычном формате ASCII.

Символьная строка определяется только директивой DB, в которой указывается более двух символов в нормальной последовательности слева направо. Следовательно, директива DB представляет единственно возможный формат для определения символьных данных.

Числовые константы

Числовые константы используются для арифметических величин и для адресов памяти. Для описания константы кавычки не ставятся. Ассемблер преобразует все числовые константы в шестнадцатеричные и записывает байты в объектном коде в обратной последовательности — справа налево. Ниже показаны различные числовые форматы.

Десятичный формат

Десятичный формат допускает десятичные цифры от 0 до 9 и обозначается последней буквой D, которую можно не указывать, например, 125 или 125D. Несмотря на то, что Ассемблер позволяет кодирование в десятичном формате, он преобразует эти значения в шест. объектный код. Например, десятичное число 125 преобразуется в шест.7D.

Шестнадцатеричный формат

Шестнадцатеричный формат допускает шест. цифры от 0 до F и обозначается последней буквой H.

Так как Ассемблер полагает, что с буквы начинаются идентификаторы, то первой цифрой шест. константы должна быть цифра от 0 до 9. Например, 2EH или 0FFFH, которые Ассемблер преобразует соответственно в 2E и FF0F (байты во втором примере записываются в объектный код в обратной последовательности).

Двоичный формат

Двоичный формат допускает двоичные цифры 0 и 1 и обозначается последней буквой B. Двоичный формат обычно используется для более четкого представления битовых значений в логических командах AND, OR, XOR и TEST. Десятичное 12, шест. C и двоичное 1100B все генерируют один и тот же код: шест. 0C или двоичное 0000 1100 в зависимости от того, как вы рассматриваете содержимое байта.

Восьмеричный формат

Восьмеричный формат допускает восьмеричные цифры от 0 до 7 и обозначается последней буквой Q или O, например, 253Q. На сегодня восьмеричный формат используется весьма редко.

Десятичный формат с плавающей точкой

Этот формат поддерживается только Ассемблером MASM. При записи символьных и числовых констант следует помнить, что, например, символьная константа, определенная как DB '12', представляет символы ASCII и генерирует шест.3132, а числовая константа, определенная как DB 12, представляет двоичное число и генерирует шест.0C.



Директива определения байта (DB)

Из различных директив, определяющих элементы данных, наиболее полезной является DB (определить байт). Символьное выражение в директиве DB может содержать строку символов любой длины, вплоть до конца строки. Объектный код показывает символы кода ASCII для каждого байта. Шест.20 представляет символ пробела.

Числовое выражение в директиве DB может содержать одну или более однобайтовых констант. Один байт выражается двумя шест. цифрами.

Наибольшее положительное шест. число в одном байте это 7F, все «большие» числа от 80 до FF представляют отрицательные значения. В десятичном исчислении эти пределы выражаются числами +127 и -128.



Директива определения слова (DW)

Директива DW определяет элементы, которые имеют длину в одно слово (два байта). Символьное выражение в DW ограничено двумя символами, которые Ассемблер представляет в объектном коде так, что, например, 'PC' становится 'CP'. Для определения символьных строк директива DW имеет ограниченное применение.

Числовое выражение в DW может содержать одно или более двухбайтовых констант. Два байта представляются четырьмя шест. цифрами. Наибольшее положительное шест. число в двух байтах это 7FFF; все «большие» числа от 8000 до FFFF представляют отрицательные значения. В десятичном исчислении эти пределы выражаются числами +32767 и -32768. Для форматов директив DW, DD и DQ Ассемблер преобразует константы в шест. объектный код, но записывает его в обратной последовательности. Таким образом десятичное значение 12345 преобразуется в шест.3039, но записывается в объектном коде как 3930.



Директива определения двойного слова (DD)

Директива DD определяет элементы, которые имеют длину в два слова (четыре байта). Числовое выражение может содержать одну или более констант, каждая из которых имеет максимум четыре байта (восемь шест. цифр).

Наибольшее положительное шест. число в четырех байтах это 7FFFFFFF; все «большие» числа от 80000000 до FFFFFFFF представляют отрицательные значения.

В десятичном исчислении эти пределы выражаются числами +2147483647 и -2147483648.

Ассемблер преобразует все числовые константы в директиве DD в шест. представление, но записывает объектный код в обратной последовательности.

Таким образом десятичное значение 12345 преобразуется в шест.00003039, но записывается в объектном коде как 39300000.

Символьное выражение директивы DD ограничено двумя символами. Ассемблер преобразует символы и выравнивает их слева в четырехбайтовом двойном слове, как показано в поле FLD2DD в объектном коде.



Директива определения учетверенного слова (DQ)

Директива DQ определяет элементы, имеющие длину четыре слова (восемь байт). Числовое выражение может содержать одну или более констант, каждая из которых имеет максимум восемь байт или 16 шест.

цифр. Наибольшее положительное шест. число — это семерка и 15 цифр F. Для получения представления о величине этого числа, покажем, что шест. 1 и 15 нулей эквивалентен следующему десятичному числу:

```
1152921504606846976
```

Ассемблер преобразует все числовые константы в директиве DQ в шест. представление, но записывает объектный код в обратной последовательности, как и в директивах DD и DW.

Обработка Ассемблером символьных строк в директиве DQ аналогична директивам DD и DW.



Директива определения десяти байт (DT)

Директива DT определяет элементы данных, имеющие длину в десять байт.

Назначение этой директивы связано с «упакованными десятичными» числовыми величинами.

По директиве DT генерируются различные константы, в зависимости от версии Ассемблера.



Непосредственные операнды

Ранее было показано использование непосредственных операндов. Команда:

```
MOV AX,0123H
```

пересылает непосредственную шест. константу 0123 в регистр AX.

Трехбайтный объектный код для этой команды есть B82301, где B8 обозначает «переслать непосредственное значение в регистр AX», а следующие два байта содержат само значение.

Многие команды имеют два операнда: первый может быть регистр или адрес памяти, а второй — непосредственная константа.

Использование непосредственного операнда более эффективно, чем определение числовой константы в сегменте данных и организация ссылки на нее в операнде команды MOV, например,

Сегмент данных: AMT1 DW 0123H

Сегмент кодов: MOV AX,AMT1

Длина непосредственных операндов

Длина непосредственной константы зависит от длины первого операнда. Например, следующий непосредственный операнд является двухбайтовым, но регистр AL имеет только один байт:

```
MOV AL,0123H (ошибка)
```

Однако, если непосредственный операнд короче, чем получающий операнд, как в следующем примере

```
ADD AX,25H (нет ошибки)
```

то Ассемблер расширяет непосредственный операнд до двух байт, 0025 и записывает объектный код в виде 2500.

Непосредственные форматы

Непосредственная константа может быть шестнадцатеричной, например, 0123H; десятичной, например, 291 (которую Ассемблер конвертирует в шест.0123); или двоичной, например, 100100011B (которая преобразуется в шест. 0123). Ниже приведен список команд, которые допускают непосредственные операнды:

Команды пересылки и сравнения

MOV, CMP.

Арифметические команды

ADC, ADD, SBB, SUB.

Команды сдвига

RCL, RCR, ROL, ROR, SHL, SAR, SHR.

Логические команды

AND, OR, TEST, XOR.

Для создания элементов, длиннее чем два байта, можно использовать цикл или строковые команды.



Директива EQU

Директива EQU не определяет элемент данных, но определяет значение, которое может быть использовано для постановки в других командах. Предположим, что в сегменте данных закодирована следующая директива EQU:

```
TIMES EQU 10
```

Имя, в данном случае TIMES, может быть представлено любым допустимым в Ассемблере именем. Теперь, в какой бы команде или директиве не использовалось слово TIMES Ассемблер подставит значение 10. Например, Ассемблер преобразует директиву

```
FIELDA DB TIMES DUP (?) в FIELDA DB 10 DUP (?)
```

Имя, связанное с некоторым значением с помощью директивы EQU, может использоваться в командах, например:

```
COUNTR EQU 05 ...
MOV CX,COUNTR
```

Ассемблер заменяет имя COUNTR в команде MOV на значение 05, создавая операнд с непосредственным значением, как если бы было закодировано:

```
MOV CX,05 ;Ассемблер подставляет 05
```

Здесь преимущество директивы EQU заключается в том, что многие команды могут использовать значение, определенное по имени COUNTR. В случае, если это значение должно быть изменено, то изменению подлежит лишь одна директива EQU. Естественно, что использование директивы EQU разумно лишь там, где подстановка имеет смысл для Ассемблера. В директиве EQU можно использовать символические имена:

```
1. TP EQU TOTALPAY
2. MPY EQU MUL
```

Первый пример предполагает, что в сегменте данных программы определено имя TOTALPAY. Для любой команды, содержащей операнд TP, Ассемблер заменит его на адрес TOTALPAY. Второй пример показывает возможность использования в программе слова MPY вместо обычного мнемокода MUL.

Лекция 3. Регистры



Сегментные регистры: CS, DS, SS и ES

Каждый сегментный регистр обеспечивает адресацию 64 К памяти, которая называется текущим сегментом. Как показано ранее, сегмент выравнен на границу параграфа и его адрес в сегментном регистре предполагает наличие справа четырех нулевых битов.

Регистр CS

Регистр сегмента кода содержит начальный адрес сегмента кода. Этот адрес плюс величина смещения в командном указателе (IP) определяет адрес команды, которая должна быть выбрана для выполнения. Для обычных программ нет необходимости делать ссылки на регистр CS.

Регистр DS

Регистр сегмента данных содержит начальный адрес сегмента данных. Этот адрес плюс величина смещения, определенная в команде, указывают на конкретную ячейку в сегменте данных.

Регистр SS

Регистр сегмента стека содержит начальный адрес в сегменте стека.

Регистр ES

Некоторые операции над строками используют дополнительный сегментный регистр для управления адресацией памяти. В данном контексте регистр ES связан с индексным регистром DI. В случае, если необходимо использовать регистр ES, ассемблерная программа должна его инициализировать.



Регистры общего назначения: AX, BX, CX и DX

При программировании на Ассемблере регистры общего назначения являются «рабочими лошадками». Особенность этих регистров состоит в том, что возможна адресация их как одного целого слова или как однобайтовой части. Левый байт является старшей частью (high), а правый — младшей частью (low). Например, двухбайтовый регистр CX состоит из двух однобайтовых: CH и CL, и ссылки на регистр возможны по любому из этих трех имен. Следующие три ассемблерные команды засылают нули в регистры CX, CH и CL, соответственно:

```
MOV CX,00
MOV CH,00
MOV CL,00
```

Регистр AX

Регистр AX является основным сумматором и применяется для всех операций ввода-вывода, некоторых операций над строками и некоторых арифметических операций. Например, команды умножения, деления и сдвига предполагают использование регистра AX.

Некоторые команды генерируют более эффективный код, если они имеют ссылки на регистр AX.

AX: | AH | AL |

Регистр BX

Регистр BX является базовым регистром. Это единственный регистр общего назначения, который может использоваться в качестве «индекса» для расширенной адресации. Другое общее применение его — вычисления.

BX: | BH | BL |

Регистр CX

Регистр CX является счетчиком. Он необходим для управления числом повторений циклов и для операций сдвига влево или вправо. Регистр CX используется также для вычислений.

CX: | CH | CL |

Регистр DX

Регистр DX является регистром данных. Он применяется для некоторых операций ввода/вывода и тех операций умножения и деления над большими числами, которые используют регистровую пару DX и AX.

DX: | DH | DL |

Любые регистры общего назначения могут использоваться для сложения и вычитания как 8-ми, так и 16-ти битовых значений.



Регистровые указатели: SP и BP

Регистровые указатели SP и BP обеспечивают системе доступ к данным в сегменте стека. Реже они используются для операций сложения и вычитания.

Регистр SP

Указатель стека обеспечивает использование стека в памяти, позволяет временно хранить адреса и иногда данные.

Этот регистр связан с регистром SS для адресации стека.

Регистр BP

Указатель базы облегчает доступ к параметрам: данным и адресам переданным через стек.



Индексные регистры: SI и DI

Оба индексных регистра возможны для расширенной адресации и для использования в операциях сложения и вычитания.

Регистр SI

Этот регистр является индексом источника и применяется для некоторых операций над строками. В данном контексте регистр SI связан с регистром DS.

Регистр DI

Этот регистр является индексом назначения и применяется также для строковых операций. В данном контексте регистр DI связан с регистром ES.



Регистр командного указателя: IP

Регистр IP содержит смещение на команду, которая должна быть выполнена. Обычно этот регистр в программе не используется, но он может изменять свое значение при использовании отладчика DOS DEBUG для тестирования программы.



Флаговый регистр

Девять из 16 битов флагового регистра являются активными и определяют текущее состояние машины и результатов выполнения. Многие арифметические команды и команды сравнения изменяют состояние флагов.

Назначение флаговых битов

O (Переполнение)

Указывает на переполнение старшего бита при арифметических командах.

D (Направление)

Обозначает левое или правое направление пересылки или сравнения строковых данных (данных в памяти превышающих длину одного слова).

I (Прерывание)

Указывает на возможность внешних прерываний.

T (Пошаговый режим)

Обеспечивает возможность работы процессора в пошаговом режиме. Например, программа DOS DEBUG устанавливает данный флаг так, что возможно пошаговое выполнение каждой команды для проверки изменения содержимого регистров и памяти.

S (Знак)

Содержит результирующий знак после арифметических операций (0 — плюс, 1 — минус).

Z (Ноль)

Показывает результат арифметических операций и операций сравнения (0 — ненулевой, 1 — нулевой результат).

A (Внешний перенос)

Содержит перенос из 3-го бита для 8-битных данных используется для специальных арифметических операций.

P (Контроль четности)

Показывает четность младших 8-битовых данных (1 — четное и 0 — нечетное число).

C (Перенос)

Содержит перенос из старшего бита, после арифметических операций, а также последний бит при сдвигах или циклических сдвигах. При программировании на Ассемблере наиболее часто используются флаги O, S, Z, и C для арифметических операций и операций сравнения, а флаг D для обозначения направления в операциях над строками.

Лекция 4. Арифметические операции



Обработка двоичных данных

Несмотря на то, что мы привыкли к десятичной арифметике (база 10), компьютер работает только с двоичной арифметикой (база 2). Кроме того, ввиду ограничения, накладываемого 16-битовыми регистрами, большие величины требуют специальной обработки.

Сложение и вычитание

Команды ADD и SUB выполняют сложение и вычитание байтов или слов, содержащих двоичные данные. Вычитание выполняется в компьютере по методу сложения с двоичным дополнением: для второго операнда устанавливаются обратные значения бит и прибавляется 1, а затем происходит сложение с первым операндом. Во всем, кроме первого шага, операции сложения и вычитания идентичны.

Поскольку прямой операции память-память не существует, данная операция выполняется через регистр. В следующем примере к содержимому слова WORDB прибавляется содержимое слова WORDA, описанных как DW:

```
MOV AX,WORDA
ADD AX,WORDB
MOV WORDB,AX
```

Переполнения

Опасайтесь переполнений в арифметических операциях. Один байт содержит знаковый бит и семь бит данных, то есть, значения от -128 до +127.

Результат арифметической операции может легко превзойти емкость однобайтового регистра. Например, результат сложения в регистре AL, превышающий его емкость, автоматически не переходит в регистр AH.

Предположим, что регистр AL содержит шест.60, тогда результат команды

```
ADD AL,20H
```

генерирует в AL сумму — шест.80. Но операция также устанавливает флаг переполнения и знаковый флаг в состояние «отрицательно». Причина заключается в том, что шест.80 или двоичное 1000 0000 является отрицательным числом, то есть, в результате, вместо +128, мы получим -128.

Так как регистр AL слишком мал для такой операции и следует воспользоваться регистром AX.

В следующем примере команда CBW (Convert Byte to Word — преобразовать байт в слово) преобразует шест.60 в регистре AL в шест.0060 в регистре AX, передавая при этом знаковый бит (0) через регистр AH.

Команда ADD генерирует теперь в регистре AX правильный результат: шест.0080, или +128:

```
CBW ;Расширение AL до AX
ADD AX,20H ;Прибавить к AX
```

Но полное слово имеет также ограничение: один знаковый бит и 15 бит данных, что соответствует значениям от -32768 до +32767.



Беззнаковые и знаковые данные

Многие числовые поля не имеют знака, например, номер абонента, адрес памяти. Некоторые числовые поля предлагаются всегда положительные, например, норма выплаты, день недели, значение числа промежуточного итога. Другие числовые поля являются знаковые, так как их содержимое может быть положительным или отрицательным. Например, долговой баланс покупателя, который может быть отрицательным при переплатах, или алгебраическое число.

Для беззнаковых величин все биты являются битами данных и вместо ограничения +32767 регистр может содержать числа до +65535. Для знаковых величин левый байт является знаковым битом.

Команды ADD и SUB не делают разницы между знаковыми и беззнаковыми величинами, они просто складывают и вычитают биты.

В следующем примере сложения двух двоичных чисел, первое число содержит единичный левый бит.

Для беззнакового числа биты представляют положительное число 249, для знакового — отрицательное число -7:

```
                          Беззнаковое    Знаковое
1111      1001            249            -7
+         +               +              +
0000      0010            2              +2
---------------------------------------------
1111      1011            251            -5
```

Двоичное представление результата сложения одинаково для беззнакового и знакового числа.

Однако, биты представляют +251 для беззнакового числа и -5 для знакового. Таким образом, числовое содержимое поля может интерпретироваться по разному.

Состояние «перенос» возникает в том случае, когда имеется перенос в знаковый разряд.

Состояние «переполнение» возникает в том случае, когда перенос в знаковый разряд не создает переноса из разрядной сетки или перенос из разрядной сетки происходит без переноса в знаковый разряд.

При возникновении переноса при сложении беззнаковых чисел, результат получается неправильный.

При возникновении переполнения при сложении знаковых чисел, результат получается неправильный.

При операциях сложения и вычитания может одновременно возникнуть и переполнение, и перенос.

Умножение

Операция умножения для беззнаковых данных выполняется командой MUL, а для знаковых — IMUL (Integer MULtiplication — умножение целых чисел).

Ответственность за контроль над форматом обрабатываемых чисел и за выбор подходящей команды умножения лежит на самом программисте. Существуют две основные операции умножения:

Байт на байт

Множимое находится в регистре AL, а множитель в байте памяти или в однобайтовом регистре. После умножения произведение находится в регистре AX. Операция игнорирует и стирает любые данные, которые находились в регистре AH.

Слово на слово

Множимое находится в регистре AX, а множитель — в слове памяти или в регистре. После умножения произведение находится в двойном слове, для которого требуется два регистра: старшая (левая) часть произведения находится в регистре DX, а младшая (правая) часть в регистре AX. Операция игнорирует и стирает любые данные, которые находились в регистре DX.

В единственном операнде команд MUL и IMUL указывается множитель. Рассмотрим следующую команду:

```
MUL MULTR
```

В случае, если поле MULTR определено как байт (DB), то операция предполагает умножение содержимого AL на значение байта из поля MULTR. В случае, если поле MULTR определено как слово (DW), то операция предполагает умножение содержимого AX на значение слова из поля MULTR. В случае, если множитель находится в регистре, то длина регистра определяет тип операции, как это показано ниже:

```
MUL CL ;Байт-множитель: множимое в AL, произвед. в AX
MUL BX ;Слово-множитель:множимое в AX, произвед. в DX:AX
```

Беззнаковое умножение: Команда MUL

Команда MUL (MULtiplication — умножение) умножает беззнаковые числа.

Знаковое умножение: Команда IMUL

Команда IMUL (Integer MULtiplication — умножение целых чисел) умножает знаковые числа.

Команда MUL рассматривает шест.80 как +128, а команда IMUL — как -128. В результате умножения -128 на +64 получается -8192 или шест.E000.

Если множимое и множитель имеет одинаковый знаковый бит, то команды MUL и IMUL генерируют одинаковый результат. Но, если сомножители имеют разные знаковые биты, то команда MUL вырабатывает положительный результат умножения, а команда IMUL — отрицательный.

Повышение эффективности умножения

При умножении на степень числа 2 (2,4,8 и так далее) более эффективным является сдвиг влево на требуемое число битов. Сдвиг более чем на 1 требует загрузки величины сдвига в регистр CL. В следующих примерах предположим, что множимое находится в регистре AL или AX:

Умножение на 2:

```
SHL AL,1
```

Умножение на 8:

```
MOV CL,3
SHL AX,CL
```

Многословное умножение

Обычно умножение имеет два типа: «байт на байт» и «слово на слово».

Как уже было показано, максимальное знаковое значение в слове ограничено величиной +32767. Умножение больших чисел требует выполнения некоторых дополнительных действий. Рассматриваемый подход предполагает умножение каждого слова отдельно и сложение полученных результатов. Рассмотрим следующее умножение в десятичном формате:

```
1365
x
12
------
 2730
1365
------
16380
```


Представим, что десятичная арифметика может умножать только двузначные числа. Тогда можно умножить 13 и 65 на 12 раздельно, следующим образом:

```
13          65
x           x
12          12
---         ---
 26         130
13          65
---         ---
156         780
```

Следующим шагом сложим полученные произведения, но поскольку число 13 представляло сотни, то первое произведение в действительности будет 15600:

```
15600
+
 780
-------
16380
```

Ассемблерная программа использует аналогичную технику за исключением того, что данные имеют размерность слов (четыре цифры) в шестнадцатеричном формате.

Умножение двойного слова на слово

Процедура E10XMUL умножает двойное слово на слово. Множимое, MULTCND, состоит из двух слов, содержащих соответственно шест.3206 и шест.2521. Определение данных в виде двух слов (DW) вместо двойного слова (DD) обусловлено необходимостью правильной адресации для команд MOV, пересылающих слова в регистр AX. Множитель MULTPLR содержит шест.6400.

Область для записи произведения, PRODUCT, состоит из трех слов. Первая команда MUL перемножает MULTPLR и правое слово поля MULTCND; произведение — шест.0E80 E400 записывается в PRODUCT+2 и PRODUCT+4. Вторая команда MUL перемножает MULTPLR и левое слово поля MULTCND, получая в результате шест. 138A 5800. Далее выполняется сложение двух произведений следующим образом:

```
Произведение 1:    0000    0E80    E400
Произведение 2:    138A    5800
-
Результат:         138A    6680    E400
```

Так как первая команда ADD может выработать перенос, то второе сложение выполняется командой сложения с переносом ADC (ADd with Carry).

В силу обратного представления байтов в словах, область PRODUCT в действительности будет содержать значение 8A13 8066 00E4. Программа предполагает, что первое слово в области PRODUCT имеет начальное значение 0000.

Умножение двойного слова на двойное слово

Умножение двух двойных слов включает следующие четыре операции умножения:

```
Множимое  Множитель
слово 2   слово 2
слово 2   слово 1
слово 1   слово 2
слово 1   слово 1
```

Каждое произведение в регистрах DX и AX складывается с соответствующим словом в окончательном результате.

Хотя логика умножения двойных слов аналогична умножению двойного слова на слово, имеется одна особенность, после пары команд сложения ADD/ADC используется еще одна команда ADC, которая прибавляет 0 к значению в итоговом поле.

Это необходимо потому, что первая команда ADC сама может вызвать перенос, который последующие команды могут стереть. Поэтому вторая команда ADC прибавит 0, если переноса нет, и прибавит 1, если перенос есть.

Финальная пара команд ADD/ADC не требует дополнительной команды ADC, так как область итога достаточно велика для генерации окончательного результата и переноса на последнем этапе не будет.



Сдвиг регистровой пары DX:AX

Следующая подпрограмма может быть полезна для сдвига содержимого регистровой пары DX:AX вправо или влево. Можно придумать

более эффективный метод, но данный пример представляет общий подход для любого числа циклов (и, соответственно, сдвигов) в регистре CX. Заметьте, что сдвиг единичного бита за разрядную сетку устанавливает флаг переноса.

Сдвиг влево на 4 бита

```
MOV CX,04 ;Инициализация на 4 цикла C20:
SHL DX,1 ;Сдвинуть DX на 1 бит влево
SHL AX,1 ;Сдвинуть AX на 1 бит влево
ADC DX,00 ;Прибавить значение переноса
LOOP C20 ;Повторить Сдвиг вправо на 4 бита
MOV CX,04 ;Инициализация на 4 цикла D20:
SHR AX,1 ;Сдвинуть AX на 1 бит вправо
SHR DX,1 ;Сдвинуть DX на 1 бит вправо
JNC D30 ;В случае, если есть перенос,
OR AH,10000000B ; то вставить 1 в AH D30:
LOOP D20 ;Повторить
```

Ниже приведен более эффективный способ для сдвига влево, не требующий организации цикла. В этом примере фактор сдвига записывается в регистр CL.

Пример написан для сдвига на 4 бита, но может быть адаптирован для других величин сдвигов:

```
MOV CL,04 ;Установить фактор сдвига
SHL DX,CL ;Сдвинуть DX влево на 4 бита
MOV BL,AH ;Сохранить AH в BL
SHL AX,CL ;Сдвинуть AX влево на 4 бита
SHL BL,CL ;Сдвинуть BL вправо на 4 бита
OR DL,BL ;Записать 4 бита из BL в DL
```



Деление

Операция деления для беззнаковых данных выполняется командой DIV, а для знаковых — IDIV. Ответственность за подбор подходящей команды лежит на программисте.

Существуют две основные операции деления:

Деление слова на байт

Делимое находится в регистре AX, а делитель — в байте памяти или а однобайтовом регистре. После деления остаток получается в регистре AH, а частное — в AL. Так как однобайтовое частное очень мало (максимально +255 (шест.FF) для беззнакового деления и +127 (шест.7F) для знакового), то данная операция имеет ограниченное использование.

Деление двойного слова на слово

Делимое находится в регистровой паре DX:AX, а делитель — в слове памяти или а регистре. После деления остаток получается в регистре DX, а частное в регистре AX. Частное в одном слове допускает максимальное значение +32767 (шест.FFFF) для беззнакового деления и +16383 (шест.7FFF) для знакового.

В единственном операнде команд DIV и IDIV указывается делитель. Рассмотрим следующую команду:

```
DIV DIVISOR
```

В случае, если поле DIVISOR определено как байт (DB), то операция предполагает деление слова на байт. В случае, если поле DIVISOR определено как слово (DW), то операция предполагает деление двойного слова на слово.

При делении, например, 13 на 3, получается результат 4 1/3. Частное есть 4, а остаток — 1. Заметим, что ручной калькулятор выдает в этом случае результат 4,333.... Значение содержит целую часть (4) и дробную часть (,333). Значение 1/3 и 333... есть дробные части, в то время как 1 есть остаток от деления.

Беззнаковое деление: Команда DIV

Команда DIV делит беззнаковые числа.

Знаковое деление: Команда IDIV

Команда IDIV (Integer DIVide) выполняет деление знаковых чисел.

Повышение производительности

При делении на степень числа 2 (2, 4, и так далее) более эффективным является сдвиг вправо на требуемое число битов. В следующих примерах предположим, что делимое находится в регистре AX:

Деление на 2:


```
SHR AX,1
```

Деление на 8:

```
MOV CL,3
SHR AX,CL
```

Переполнения и прерывания

Используя команды DIV и особенно IDIV, очень просто вызвать переполнение. Прерывания приводят (по крайней мере в системе, используемой при тестировании этих программ) к непредсказуемым результатам. В операциях деления предполагается, что частное значительно меньше, чем делимое.

Деление на ноль всегда вызывает прерывание. Но деление на 1 генерирует частное, которое равно делимому, что может также легко вызвать прерывание.

Рекомендуется использовать следующее правило: если делитель — байт, то его значение должно быть меньше, чем левый байт (AH) делителя: если делитель — слово, то его значение должно быть меньше, чем левое слово (DX) делителя.

При использовании команды IDIV необходимо учитывать тот факт, что либо делимое, либо делитель могут быть отрицательными, а так как сравниваются абсолютные значения, то необходимо использовать команду NEG для временного перевода отрицательного значения в положительное.

Деление вычитанием

В случае, если частное слишком велико, то деление можно выполнить с помощью циклического вычитания.

Метод заключается в том, что делитель вычитается из делимого и в этом же цикле частное увеличивается на 1.

Вычитание продолжается, пока делимое остается больше делителя. В следующем примере, делитель находится в регистре AX, а делимое — в BX, частное вырабатывается в CX:

```
SUB CX,CX ;Очистка частного C20:
CMP AX,BX ;В случае, если делимое < делителя,
JB C30 ; то выйти
SUB AX,BX ;Вычитание делителя из делимого
INC CX ;Инкремент частного
JMP C20 ;Повторить цикл C30:
RET ;Частное в CX, остаток в AX
```

В конце подпрограммы регистр CX будет содержать частное, а AX — остаток. Пример умышленно примитивен для демонстрации данной техники деления. В случае, если частное получается в регистровой паре DX:AX, то необходимо сделать два дополнения:

1. В метке C20 сравнивать AX и BX только при нулевом DX.

2. После команды SUB вставить команду SBB DX,00.

Примечание: очень большое частное и малый делитель могут вызвать тысячи циклов.



Преобразование знака

Команда NEG обеспечивает преобразование знака двоичных чисел из положительного в отрицательное и наоборот. Практически команда NEG устанавливает противоположные значения битов и прибавляет 1. Примеры:

```
NEG AX
NEG BL
NEG BINAMT ;(байт или слово в памяти)
```

Преобразование знака для 35-битового (или большего) числа включает больше шагов. Предположим, что регистровая пара DX:AX содержит 32-битовое двоичное число. Так как команда NEG не может обрабатывать два регистра одновременно, то ее использование приведет к неправильному результату. В следующем примере показано использование команды NOT:

```
NOT DX ;Инвертирование битов
NOT AX ;Инвертирование битов
ADD AX,1 ;Прибавление 1 к AX
ADC DX,0 ;Прибавление переноса к DX
```

Остается одна незначительная проблема: над числами, представленными в двоичном формате, удобно выполнять арифметические операции, если сами числа определены в программе. Данные, вводимые в программу с дискового файла, могут также иметь двоичный формат. Но данные, вводимые с клавиатуры, представлены в ASCII-формате.

Хотя ASCII-коды удобны для отображения и печати, они требуют специальных преобразований в двоичный формат для арифметических вычислений.

Важно:

- Будьте особенно внимательны при использовании однобайтовых регистров. Знаковые значения здесь могут быть от -128 до +127.
- Для многословного сложения используйте команду ADC для учета переносов от предыдущих сложений. В случае, если операция выполняется в цикле, то используя команду CLC, установите флаг переноса в 0.
- Используйте команды MUL или DIV для беззнаковых данных и команды IMUL или IDIV для знаковых.
- При делении будьте осторожны с переполнениями. В случае, если нулевой делитель возможен, то обеспечьте проверку этой операции. Кроме того, делитель должен быть больше содержимого регистра AH (для байта) или DX (для слова).
- Для умножения или деления на степень двойки используйте сдвиг. Сдвиг вправо выполняется командой SHR для беззнаковых полей и командой SAR для знаковых полей. Для сдвига влево используются идентичные команды SHL и SAL.
- Будьте внимательны при ассемблировании по умолчанию. Например, если поле FACT определено как байт (DB), то команда MUL FACT полагает множимое в регистре AL, а команда DIV FACT полагает делимое в регистре AX. В случае, если FACT определен как слово (DW), то команда MUL FACT полагает множимое в регистре AX, а команда DIV FACT полагает делимое в регистровой паре DX:AX.



Обработка данных в форматах ASCII и BCD

Для получения высокой производительности компьютер выполняет арифметические операции над числами в двоичном формате. Этот формат не вызывает особых трудностей, если данные определены в самой программе. Во многих случаях новые данные вводятся программой с клавиатуры в виде ASCII символов в десятичном формате. Аналогично вывод информации на экран осуществляется в кодах ASCII. Например, число 23 в двоичном представлении выглядит как 00010111 или шест.17; в коде ASCII на каждый символ требуется один байт и число 25 в ASCII-коде имеет внутреннее представление шест.3235.

При программировании на языках высокого уровня для обозначения порядка числа или положения десятичной запятой (точки) можно положиться на компилятор. Однако, компьютер не распознает десятичную запятую (точку) в арифметических полях. Так как двоичные числа не имеют возможности установки десятичной (или двоичной) запятой (точки), то именно программист должен подразумевать и определить порядок обрабатываемых чисел.

ASCII-формат

Данные, вводимые с клавиатуры, имеют ASCII-формат, например, буквы SAM имеют в памяти шестнадцатеричное представление 53414D, цифры 1234 — шест.31323334. Во многих случаях формат алфавитных данных, например, имя человека или описание статьи, не меняется в программе. Но для выполнения арифметических операций над числовыми значениями, такими как шест.31323334, требуется специальная обработка.

С помощью следующих ассемблерных команд можно выполнять арифметические операции непосредственно над числами в ASCII-формате:

```
AAA (ASCII Adjust for Addition – коррекция для сложения
ASCII-кода)
AAD (ASCII Adjust for Division – коррекция для деления
ASCII-кода)
AAM (ASCII Adjust for Multiplication – коррекция для умножения
ASCII-кода)
```



```
AAS (ASCII Adjust for Subtraction – коррекция для вычитания
ASCII-кода)
```

Эти команды кодируются без операндов и выполняют автоматическую коррекцию в регистре AX. Коррекция необходима, так как ASCII-код представляет так называемый распакованный десятичный формат, в то время, как компьютер выполняет арифметические операции в двоичном формате.

Сложение многобайтовых ASCII-чисел требует организации цикла, который выполняет обработку справа налево с учетом переноса.

Вычитание в ASCII-формате

Команда AAS (ASCII Adjust for Subtraction — коррекция для вычитания ASCII-кодов) выполняется аналогично команде AAA. Команда AAS проверяет правую шест. цифру (четыре бита) в регистре AL. В случае, если эта цифра лежит между A и F или флаг AF равен 1, то из регистра AL вычитается 6, а из регистра AH вычитается 1, флаги AF и CF устанавливаются в 1. Во всех случаях команда AAS устанавливает в 0 левую шест.цифру в регистре AL.

Умножение в ASCII-формате

Команда AAM (ASCII Adjust for Multiplication — коррекция для умножения ASCII-кодов) выполняет корректировку результата умножения ASCII-кодов в регистре AX. Однако, шест. цифры должны быть очищены от троек и полученные данные уже не будут являться действительными ASCII-кодами. Например, число в ASCII-формате 31323334 имеет распакованное десятичное представление 01020304. Кроме этого, надо помнить, что коррекция осуществляется только для одного байта за одно выполнение, поэтому можно умножать только одно-байтовые поля; для более длинных полей необходима организация цикла.

Команда AAM делит содержимое регистра AL на 10 (шест.0A) и записывает частное в регистр AH, а остаток в AL. Предположим, что в регистре AL содержится шест.35, а в регистре CL — шест.39. Следующие команды умножают содержимое регистра AL на содержимое CL и преобразуют результат в ASCII-формат:

```
AX: AND CL,0FH ;Преобразовать CL в 09
AND AL,0FH ;Преобразовать AL в 05 0005
MUL CL ;Умножить AL на CL 002D
AAM ;Преобразовать в распак.дес. 0405
OR AX,3030H ;Преобразовать в ASCII-ф-т 3435
```

Команда MUL генерирует 45 (шест.002D) в регистре AX, после чего команда AAM делит это значение на 10, записывая частное 04 в ре-

гистр AH и остаток 05 в регистр AL. Команда OR преобразует затем распакованное десятичное число в ASCII-формат.

Деление в ASCII-формате

Команда AAD (ASCII Adjust for Division — коррекция для деления ASCII-кодов) выполняет корректировку ASCII-кода делимого до непосредственного деления. Однако, прежде необходимо очистить левые тройки ASCII-кодов для получения распакованного десятичного формата. Команда AAD может оперировать с двухбайтовыми делимыми в регистре AX. Предположим, что регистр AX содержит делимое 3238 в ASCII-формате и регистр CL содержит делитель 37 также в ASCII-формате. Следующие команды выполняют коррекцию для последующего деления:

```
AX: AND CL,0FH ;Преобразовать CL в распак.дес.
AND AX,0F0FH ;Преобразовать AX в распак.дес. 0208
AAD ;Преобразовать в двоичный 001C
DIV CL ;Разделить на 7 0004
```

Команда AAD умножает содержимое AH на 10 (шест.0A), прибавляет результат 20 (шест.14) к регистру AL и очищает регистр AH. Значение 001C есть шест. представление десятичного числа 28. Делитель может быть только однобайтовый от 01 до 09.



Двоично-десятичный формат (BCD)

В предыдущем примере деления в ASCII-формате было получено частное 00090204. В случае, если сжать это значение, сохраняя только правые цифры каждого байта, то получим 0924. Такой формат называется двоично-десятичным (BCD — Binary Coded Decimal) (или упакованным). Он содержит только десятичные цифры от 0 до 9. Длина двоично-десятичного представления в два раза меньше ASCII-представления.

Заметим, однако, что десятичное число 0924 имеет основание 10 и, будучи преобразованным в основание 16 (то есть, в шест. представление), даст шест.039C.

Можно выполнять сложение и вычитание чисел в двоично-десятичном представлении (BCD-формате).

Для этих целей имеются две корректирующих команды:

- DAA (Decimal Adjustment for Addition — десятичная коррекция для сложения)
- DAS (Decimal Adjustment for Subtraction — десятичная коррекция для вычитания)

Обработка полей также осуществляется по одному байту за одно выполнение.

Преобразование ASCII-формата в двоичный формат

Выполнение арифметических операций над числами в ASCII или BCD форматах удобно лишь для коротких полей. В большинстве случаев для арифметических операций используется преобразование в двоичный формат.

Практически проще преобразование из ASCII-формата непосредственно в двоичный формат, чем преобразование из ASCII- в BCD-формат и, затем, в двоичный формат: Метод преобразования базируется на том, что ASCII-формат имеет основание 10, а компьютер выполняет арифметические операции только над числами с основанием 2. Процедура преобразования заключается в следующем:

1. Начинают с самого правого байта числа в ASCII-формате и обрабатывают справа налево.

2. Удаляют тройки из левых шест.цифр каждого ASCII-байта.

3. Умножают ASCII-цифры на 1, 10, 100 (шест.1, A, 64) и так далее и складывают результаты.

Преобразование двоичного формата в ASCII-формат

Для того, чтобы напечатать или отобразить на экране арифметический результат, необходимо преобразовать его в ASCII-формат. Данная операция включает в себя процесс обратный предыдущему. Вместо умножения используется деление двоичного числа на 10 (шест.0A) пока результат не будет меньше 10. Остатки, которые лежат в границах от 0 до 9, образуют число в ASCII-формате.



Сдвиг и округление

Рассмотрим процесс округления числа до двух десятичных знаков после запятой. В случае, если число равно 12,345, то необходимо прибавить 5 к отбрасываемому разряду и сдвинуть число вправо на один десятичный разряд:

```
Число: 12,345 Плюс 5: +5 – Округленное число: 12,350 = 12,35
```

В случае, если округляемое число равно 12,3455, то необходимо прибавить 50 и сдвинуть на два десятичных разряда. Для 12,34555 необходимо прибавить 500 и сдвинуть на три десятичных разряда:

```
12,3455 12,34555 +50 +500 --- 12,3505 = 12,35 12,35055 = 12,35
```

К числу, имеющему шесть знаков после запятой, необходимо прибавить 5000 и сдвинуть на четыре десятичных разряда и так далее. Поскольку данные представляются в компьютере в двоичном виде, то 12345 выглядит как шест.3039. Прибавляя 5 к 3039, получим 303E, что соответствует числу 12350 в десятичном представлении. Пока все хорошо. Но вот сдвиг на одну двоичную цифру дает в результате шест.181F, или 1675 — то есть, сдвиг на одну двоичную цифру просто делит число пополам. Но нам необходим такой сдвиг, который эквивалентен сдвигу вправо на одну десятичную цифру. Такой сдвиг можно осуществить делением на 10 (шест.A):

```
Шест.303E : Шест.A = 4D3 или дес.1235
```

Преобразование шест.4D3 в ASCII-формат дает число 1235. Теперь остается лишь вставить запятую в правильную позицию числа 12,35, и можно выдать на экран округленное и сдвинутое значение.

Таким образом можно округлять и сдвигать любые двоичные числа.

Для трех знаков после запятой необходимо прибавить 5 и разделить на 10, для четырех знаков после запятой: прибавить 50 и разделить на 100. Возможно вы заметили модель: фактор округления (5, 50, 500 и так далее) всегда составляет половину фактора сдвига (10, 100, 1000 и так далее).

Конечно, десятичная запятая в двоичном числе только подразумевается.

Отрицательные величины

Некоторые применения программ допускают наличие отрицательных величин. Знак минус может устанавливаться после числа, например, 12,34-, или перед числом -12,34. Программа может проверять наличие минуса при преобразовании в двоичный формат. Можно оставить двоичное число положительным, но установить соответствующий индикатор исходной отрицательной величины. После завершения арифметических операций знак минус при необходимости может быть вставлен в ASCII поле.

В случае, если необходимо, чтобы двоичное число было также отрицательным, то можно преобразовать, как обычно, ASCII-формат в двоичный. Будьте внимательны при использовании команд IMUL и IDIV для обработки знаковых данных. Для округления отрицательных чисел следует не прибавлять, а вычитать фактор 5.

Важно:

- ASCII-формат требует один байт на каждый символ. В случае, если поле содержит только цифры от 0 до 9, то замена старших троек в каждом байте на нули создает распакованный десятичный формат. Сжатие числа до двух цифр в байте создает упакованный десятичный формат.
- После ASCII-сложения необходимо выполнить коррекцию с помощью команды AAA; после ASCII-вычитания — коррекция с помощью команды AAS.
- Прежде чем выполнить ASCII-умножение, необходимо преобразовать множимое и множитель в «распакованный десятичный» формат, обнулив в каждом байте левые тройки. После умножения необходимо выполнить коррекцию результата с помощью команды AAM.
- Прежде чем выполнить ASCII-деление, необходимо: 1) преобразовать делимое и делитель в «распакованный десятичный» формат, обнулив в каждом байте левые тройки и 2) выполнить коррекцию делимого с помощью команды AAD.
- Для большинства арифметических операций используйте преобразование чисел из ASCII-формата в двоичной формат. В процессе такого преобразования проверяйте на корректность ASCII-символы: они должны быть от шест.30 до шест.39, могут содержать десятичную запятую (точку) и, возможно, знак минус.

Лекция 5. Команды обработки строк



Свойства операций над строками

Часто бывает необходимо переслать или сравнить поля данных, которые превышают по длине одно слово.

Например, необходимо сравнить описания или имена для того, чтобы отсортировать их в восходящей последовательности. Элементы такого формата известны как строковые данные и могут являться как символьными, так и числовыми. Для обработки строковых данных Ассемблер имеет пять команд обработки строк:

- **MOVS** — переслать один байт или одно слово из одной области памяти в другую;
- **LODS** — загрузить из памяти один байт в регистр AL или одно слово в регистр AX;
- **STOS** — записать содержимое регистра AL или AX в память;
- **CMPS** — сравнить содержимое двух областей памяти, размером в один байт или в одно слово;
- **SCAS** — сравнить содержимое регистра AL или AX с содержимым памяти.

Префикс **REP** позволяет этим командам обрабатывать строки любой длины.

Цепочечная команда может быть закодирована для повторяющейся обработки одного байта или одного слова за одно выполнение. Например, можно выбрать «байтовую» команду для обработки строки с нечетным числом байт или «двухбайтовую» команду для обработки четного числа байт.

Например, можно кодировать операнды для команды MOVS, но опустить их для MOVSB и MOVSW. Эти команды предполагают, что регистры DI и SI содержат относительные адреса, указывающие на необходимые области памяти (для загрузки можно использовать команду LEA). Регистр SI обычно связан с регистром сегмента данных — DS:SI. Регистр DI всегда связан с регистром дополнительного сегмента — ES:DI. Следовательно, команды MOVS, STOS, CMPS и SCAS требуют инициализации регистра ES (обычно адресом в регистре DS).



REP: Префикс повторения цепочечной команды

Несмотря на то, что цепочечные команды имеют отношение к одному байту или одному слову, префикс REP обеспечивает повторение команды несколько раз. Префикс кодируется непосредственно перед цепочечной командой, например, REP MOVSB. Для использования префикса REP необходимо установить начальное значение в регистре CX. При выполнении цепочечной команды с префиксом REP происходит уменьшение на 1 значения в регистре CX до нуля.

Таким образом, можно обрабатывать строки любой длины.

Флаг направления определяет направление повторяющейся операции:

- для направления слева направо необходимо с помощью команды CLD установить флаг DF в 0;
- для направления справа налево необходимо с помощью команды STD установить флаг DF в 1.

В следующем примере выполняется пересылка 20 байт из STRING1 в STRING2. Предположим, что оба регистра DS и ES инициализированы адресом сегмента данных:

```
STRING1 DB 20 DUP('*')
STRING2 DB 20 DUP(' ') ...
CLD ;Сброс флага DF
MOV CX,20 ;Счетчик на 20 байт
LEA DI,STRING2 ;Адрес области "куда"
```

```
LEA SI,STRING1 ;Адрес области "откуда"
REP MOVSB ;Переслать данные
```

При выполнении команд CMPS и SCAS возможна установка флагов состояния, так чтобы операция могла прекратиться сразу после обнаружения необходимого условия. Ниже приведены модификации префикса REP для этих целей:

- REP — повторять операцию, пока CX не равно 0;
- REPZ или REPE — повторять операцию, пока флаг ZF показывает «равно или ноль».
- Прекратить операцию при флаге ZF, указывающему на не равно или не ноль или при CX равном 0;
- REPNE или REPNZ — повторять операцию, пока флаг ZF показывает «не равно или не ноль».
- Прекратить операцию при флаге ZF, указывающему на «равно или нуль» или при CX равным 0.



MOVS: Пересылка строк

Команда MOVS с префиксом REP и длиной в регистре CX может выполнять пересылку любого числа символов. Для области, принимающей строку, сегментным регистром, является регистр ES, а регистр DI содержит относительный адрес области, передающей строку. Сегментным регистром является регистр DS, а регистр SI содержит относительный адрес. Таким образом, в начале программы перед выполнением команды MOVS необходимо инициализировать регистр ES вместе с регистром DS, а также загрузить требуемые относительные адреса полей в регистры DI и SI.

В зависимости от состояния флага DF команда MOVS производит увеличение или уменьшение на 1 (для байта) или на 2 (для слова) содержимого регистров DI и SI. Приведем команды, эквивалентные цепочечной команде REP MOVSB:

```
JCXZ LABEL2
LABEL1: MOV AL,[SI]
MOV [DI],AL
```



```
INC/DEC DI ;Инкремент или декремент
UNC/DEC SI ;Инкремент или декремент
LOOP LABEL1
LABEL2: ...
```



LODS: Загрузка строки

Команда LODS загружает из памяти в регистр AL один байт или в регистр AX одно слово. Адрес памяти определяется регистрами DS:SI. В зависимости от значения флага DF происходит увеличение или уменьшение регистра SI.

Поскольку одна команда LODS загружает регистр, то практической пользы от префикса REP в данном случае нет. Часто простая команда MOV полностью адекватна команде LODS, хотя MOV генерирует три байта машинного кода, а LODS — только один, но требует инициализацию регистра SI. Можно использовать команду LODS в том случае, когда требуется продвигаться вдоль строки (по байту или по слову), проверяя загружаемый регистр на конкретное значение.

Команды, эквивалентные команде LODSB:

```
MOV AL,[SI]
INC SI
```



STOS: Запись строки

Команда STOS записывает (сохраняет) содержимое регистра AL или AX в байте или в слове памяти. Адрес памяти всегда представляется регистрами ES:DI. В зависимости от флага DF команда STOS также увеличивает или уменьшает адрес в регистре DI на 1 для байта или на 2 для слова.

Практическая польза команды STOS с префиксом REP — инициализация области данных конкретным значением, например, очистка

дисплейного буфера пробелами. Длина области (в байтах или в словах) загружается в регистр AX.

Команды, эквивалентные команде REP STOSB:

```
JCXZ LABEL2
LABEL1: MOV [DI],AL
INC/DEC DI ;Инкремент или декремент
LOOP LABEL1
LABEL2: ...
```



CMPS: Сравнение строк

Команда CMPS сравнивает содержимое одной области памяти (адресуемой регистрами DS:SI) с содержимыми другой области (адресуемой как ES:DI). В зависимости от флага DF команда CMPS также увеличивает или уменьшает адреса в регистрах SI и DI на 1 для байта или на 2 для слова. Команда CMPS устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса REP в регистре CX должна находиться длина сравниваемых полей. Команда CMPS может сравнивать любое число байт или слов.

Рассмотрим процесс сравнения двух строк, содержащих имена JEAN и JOAN. Сравнение побайтно слева направо приводит к следующему:

```
J : J    Равно
E : O    Не равно (E меньше O)
A : A    Равно
N : N    Равно
```

Сравнение всех четырех байт заканчивается сравнением N:N — равно/нуль. Так как имена «не равны», операция должна прекратиться, как только будет обнаружено условие «не равно».

Для этих целей команда REP имеет модификацию REPE, которая повторяет сравнение до тех пор, пока сравниваемые элементы равны, или регистр CX не равен нулю. Кодируется повторяющееся однобайтовое сравнение следующим образом:

```
REPE CMPSB
```




SCAS: Сканирование строк

Команда SCAS отличается от команды CMPS тем, что сканирует (просматривает) строку на определенное значение байта или слова. Команда SCAS сравнивает содержимое области памяти (адресуемой регистрами ES:DI) с содержимым регистра AL или AX. В зависимости от значения флага DF команда SCAS также увеличивает или уменьшает адрес в регистре DI на 1 для байта или на 2 для слова. Команда SCAS устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса REP и значения длины в регистре CX команда SCAS может сканировать строки любой длины.

Команда SCAS особенно полезна, например, в текстовых редакторах, где программа должна сканировать строки, выполняя поиск знаков пунктуации: точек, запятых и пробелов.

Команда SCASW сканирует в памяти слово на соответствие значению в регистре AX. При использовании команд LODSW или MOV для пересылки слова в регистр AX, следует помнить, что первый байт будет в регистре AL, а второй байт — в регистре AH. Так как команда SCAS сравнивает байты в обратной последовательности, то операция корректна.



Сканирование и замена

В процессе обработки текстовой информации может возникнуть необходимость замены определенных символов в тексте на другие, например, подстановка пробелов вместо различных редактирующих символов. В приведенном ниже фрагменте программы осуществляется сканирование строки STRING и замена символа амперсанд (&) на символ пробела.

Когда команда SCASB обнаружит символ & (в примере это будет позиция STRING+8), то операция сканирования прекратится и регистр DI будет содержать адрес STRING+9. Для получения адреса символа &

необходимо уменьшить содержимое DI на единицу и записать по полученному адресу символ пробела.

```
STRLEN EQU 15 ;Длина поля
STRING STRING DB 'The time&is now' ...
CLD MOV AL,'&' ;Искомый символ
MOV CX,STRLEN ;Длина поля
STRING LEA DI,STRING ;Адрес поля
STRING REPNE SCASB ;Сканировать
JNZ K20 ;Символ найден?
DEC DI ;Да - уменьшить адрес
MOV BYTE PTR[DI],20H ;Подставить пробел
K20: RET
```



Альтернативное кодирование

При использовании команд MOVSB или MOVSW Ассемблер предполагает наличие корректной длины строковых данных и не требует кодирования операндов в команде. Для команды MOVS длина должна быть закодирована в операндах. Например, если поля FLDA и FLDB определены как байтовые (DB), то команда REP MOVS FLDA,FLDB предполагает повторяющуюся пересылку байтов из поля FLDB в поле FLDA. Эту команду можно записать также в следующем виде:

```
REP MOVS ES:BYTE PTR[DI],DS:[SI]
```

Однако загрузка регистров DI и SI адресами FLDA и FLDB обязательна в любом случае.



Дублирование образца

Команда STOS бывает полезна для установки в некоторой области определенных значений байтов и слов. Для дублирования образца, длина которого превышает размер слова, можно использовать команду MOVS с небольшой модификацией. Предположим, что необходимо сформировать строку следующего вида:


```
***---***---***---***---***---
...
```

Вместо того, чтобы определять полностью всю строку, можно определить только первые шесть байтов. Закодируем образец непосредственно перед обрабатываемой строкой следующим образом:

```
PATTERN DB '***---'
DISAREA DB 42 DUP(?)
.
.
CLD
MOV CX,21
LEA DI,DISAREA
LEA SI,PATTERN
REP MOVSW
```

В процессе выполнения команда MOVSW сначала пересылает первое слово (**) из образца PATTERN в первое слово области DISAREA, затем — второе слово (*-), потом третье (--).

К этому моменту регистр DI будет содержать адрес DISAREA+6, а регистр SI — PATTERN+6, который также является адресом DISAREA. Затем команда MOVSW автоматически дублирует образец, пересылая первое слово из DISAREA в DISAREA+6, из DISAREA+2, в DISAREA+8, из DISAREA+4 в DISAREA+10 и так далее. В результате образец будет полностью продублирован по всей области DISAREA.

Данную технику можно использовать для дублирования в области памяти любого образца любой длины. Образец должен быть расположен непосредственно перед принимающей областью.

Важно:

- Для цепочечных команд MOVS, STOS, CMPS и SCAS не забывайте инициализировать регистр ES.
- Сбрасывайте (CLD) или устанавливайте (STD) флаг направления в соответствии с направлением обработки.
- Не забывайте устанавливать в регистрах DI и SI необходимые значения. Например, команда MOVS предполагает операнды DI,SI, а команда CMPS — SI,DI.

- Инициализируйте регистр CX в соответствии с количеством байтов или слов, участвующих в процессе обработки.
- Для обычной обработки используйте префикс REP для команд MOVS и STOS и модифицированный префикс (REPE или REPNE) для команд CMPS и SCAS.
- Помните об обратной последовательности байтов в сравниваемых словах при выполнении команд CMPSW и SCASW.
- При обработке справа налево устанавливайте начальные адреса на последний байт обрабатываемой области. В случае, если, например, поле NAME1 имеет длину 10 байтов, то для побайтовой обработки данных в этой области справа налево начальный адрес, загружаемый командой LEA, должен быть NAME1+9. Для обработки слов начальный адрес в этом случае — NAME1+8.

Лекция 6. Обработка таблиц



Определение таблиц

Многие программные применения используют табличную организацию таких данных, как имена, описания, размеры, цены. Определение и использование таблиц включает одну новую команду Ассемблера — XLAT. Таким образом, использование таблиц — это лишь дело техники и применения знаний, полученных из предыдущих глав.

Организация поиска в таблице зависит от способа ее определения. Существует много различных вариантов определения таблиц и алгоритмов поиска.

Для облегчения табличного поиска большинство таблиц определяются систематично, то есть, элементы таблицы имеют одинаковый формат (символьный или числовой), одинаковую длину и восходящую или нисходящую последовательность элементов.

Возьмем, к примеру, стек, представляющий собой таблицу из 64-х неинициализированных слов:

```
STACK DW 64 DUP(?)
```

Следующие две таблицы инициализированы символьными и числовыми значениями:

```
MONTAB DB 'JAN','FEB','MAR', ... ,'DEC'
COSTAB DB 205,208,209,212,215,224,...
```

Таблица MONTAB определяет алфавитные аббревиатуры месяцев, а COSTAB — определяет таблицу номеров служащих. Таблица может также содержать смешанные данные (регулярно чередующиеся числовые и символьные поля). В следующей ассортиментной таблице каждый числовой элемент (инвентарный номер) имеет две цифры (один байт), а каждый символьный элемент (наименование) имеет девять бай-

тов. Точки, показанные в наименовании «Paper» дополняют длину этого поля до 9 байт. Точки показывают, что недостающее пространство должно присутствовать. Вводить точки необязательно.

```
STOKTBL DB 12,'Computers',14,'Paper....',17,'Diskettes'
```

Для ясности можно закодировать элементы таблицы вертикально:

```
STOKTBL DB 12, 'Computers' DB 14, 'Paper....' DB 17, 'Diskettes'
```

Рассмотрим теперь различные способы использования таблиц в программах.



Прямой табличный доступ

Предположим, что пользователь ввел номер месяца — 03 и программа должна преобразовать этот номер в алфавитное значение March. Программа для выполнения такого преобразования включает определение таблицы алфавитных названий месяцев, имеющих одинаковую длину. Так как самое длинное название — September, то таблица имеет следующий вид:

```
MONTBL DB 'January..' DB 'February.' DB 'March....'
```

Каждый элемент таблицы имеет длину 9 байт. Адрес элемента 'January' — MONTBL+0, 'February' — MONTBL+9, 'March' — MONTBL+18. Для локализации месяца 03, программа должна выполнить следующее:

1. Преобразовать введенный номер месяца из ASCII 33 в двоичное 03.

2. Вычесть единицу из номера месяца: 03 — 1 = 02 3. Умножить результат на длину элемента (9): 02 x 9 = 18 4. Прибавить произведение (18) к адресу MONTBL; в результате получится адрес требуемого названия месяца: MONTBL+18.

Описанная техника работы с таблицей называется прямым табличным доступом. Поскольку данный алгоритм непосредственно вычисляет адрес необходимого элемента в таблице, то в программе не требуется выполнять операции поиска.

Хотя прямая табличная адресация очень эффективна, она возможна только при последовательной организации. То есть можно использовать такие таблицы, если элементы располагаются в регулярной последовательности: 1, 2, 3,... или 106, 107, 108,... или даже 5, 10, 15. Однако, не всегда таблицы построены таким образом.



Табличный поиск

Некоторые таблицы состоят из чисел, не имеющих видимой закономерности. Характерный пример — таблица инвентарных номеров с последовательными номерами, например, 134, 138, 141, 239 и 245. Другой тип таблиц состоит из распределенных по ранжиру величин, таких как подоходный налог.

Таблицы с уникальными элементами

Инвентарные номера большинства фирм часто не имеют последовательного порядка. Номера, обычно, группируются по категориям, первые цифры указывают на мебель или приборы, или номер отдела. Кроме того, время от времени номера удаляются, а новые добавляются. В таблице необходимо связать инвентарные номера и их конкретные наименования (и, если требуется, включить стоимость). Инвентарные номера и наименования могут быть определены в различных таблицах, например:

```
STOKNOS DB '101','107','109',...
STOKDCR DB 'Excavators','Processors','Assemblers',...
```

или в одной таблице, например:

```
STOKTAB DB '101','Excavators' DB '107','Processors'
DB '109','Assemblers' ...
```

Таблицы с ранжированием

Подоходный налог дает характерный пример таблицы с ранжированными значениями. Представим себе таблицу, содержащую размеры доходов облагаемых налогами, процент налога и поправочный коэффициент:

В налоговой таблице процент увеличивается в соответствии с увеличением налогооблагаемого дохода. Элементы таблицы доходов содержат максимальные величины для каждого шага:

```
TAXTBL DD 100000,250000,425000,600000,999999
```

Для организации поиска в такой таблице, программа сравнивает доход налогоплательщика с табличным значением дохода:

- если меньше или равно, то использовать соответствующий процент и поправку;
- если больше, то перейти к следующему элементу таблицы.

Таблицы с элементами переменной длины

Существуют таблицы, в которых элементы имеют переменную длину. Каждый элемент такой таблицы может завершаться специальным символом ограничителем, например, шест.00; конец таблицы можно обозначить ограничителем шест.FF. В этом случае необходимо гарантировать, чтобы внутри элементов таблицы не встречались указанные ограничители. Помните, что двоичные числа могут выражаться любыми битовыми комбинациями. Для поиска можно использовать команду SCAS.



Транслирующая команда XLAT

Команда XLAT транслирует содержимое одного байта в другое предопределенное значение. С помощью команды XLAT можно проверить корректность содержимого элементов данных. При передаче данных между персональным компьютером и ЕС ЭВМ (IBM) с помощью команды XLAT можно выполнить перекодировку данных между форматами ASCII и EBCDIC.

В следующем примере происходит преобразование цифр от 0 до 9 из кода ASCII в код EBCDIC. Так как представление цифр в ASCII выглядит как шест.30-39, а в EBCDIC — шест.F0-F9, то замену можно выполнить командой OR. Однако, дополнительно преобразуем все остальные коды ASCII в пробел (шест.40) в коде EBCDIC. Для команды XLAT необходимо определить таблицу перекодировки, которая учитывает все 256 возможных символов, с кодами EBCDIC в ASCII позициях:

```
XLTBL DB 47 DUP(40H) ;Пробелы в коде EBCDIC
DB 0F0H,0F1H,0F2H,0F3H,...,0F9H ;0-9 (EBCDIC)
DB 199 DUP(40H) ;Пробелы в коде EBCDIC
```


Команда XLAT предполагает адрес таблицы в регистре BX, а транслируемый байт (например, поля ASCNO) в регистре AL. Следующие команды выполняют подготовку и трансляцию байта:

```
LEA BX,XLTBL
MOV AL,ASCNO
XLAT
```

Команда XLAT использует значение в регистре AL в качестве относительного адреса в таблице, то есть, складывает адрес в BX и смещение в AL. В случае, если, например, ASCNO содержит 00, то адрес байта в таблице будет XLTBL+00 и команда XLAT заменит 00 на шест.40 из таблицы. В случае, если поле ASCNO содержит шест.32, то адрес соответствующего байта в таблице будет XLTBL+50. Этот байт содержит шест.F2 (2 в коде EBCDIC), который команда XLAT загружает в регистр AL.



Операторы типа, длина и размеры

Ассемблер содержит ряд специальных операторов, которые могут оказаться полезными при программировании. Например, при изменении длины таблицы придется модифицировать программу (для нового определения таблицы) и процедуры, проверяющие конец таблицы. В этом случае использование операторов TYPE (тип), LENGTH (длина) и SIZE (размер) позволяют уменьшить число модифицируемых команд.

Рассмотрим определение следующей таблицы из десяти слов:

```
TABLEX DW 10 DUP(?) ;Таблица из 10 слов
```

Программа может использовать оператор TYPE для определения типа (DW в данном случае), оператор LENGTH для определения DUP-фактора (10) и оператор SIZE для определения числа байтов (10 x 2 = 20). Следующие команды иллюстрируют три таких применения:

```
MOV AX,TYPE
TABLEX ;AX=0002
MOV BX,LENGTH
TABLEX ;BX=000A (10)
MOV CX,SIZE
TABLEX ;CX=0014 (20)
```

Значения LENGTH и SIZE можно использовать для окончания табличного поиска или сортировки. Например, если регистр SI содержит продвинутый адрес таблицы при осуществлении поиска, то проверка на конец таблицы может быть следующий:

```
CMP SI,SIZE
TABLEX
```

Важно:

- Для большинства применений, определяйте таблицы, имеющие родственные элементы одной длины и формата данных.
- Стройте таблицы на основе форматов данных. Например, элементы могут быть символьные или числовые длиной один, два и более байтов каждый. Может оказаться более практичным определение двух таблиц: одна, например, для трехсимвольных значений номеров, а другая для двухбайтовых значений цен единиц товара. В процессе поиска адрес элементов таблицы номеров должен увеличиваться на 3, а адрес элементов таблицы цен — на 2. В случае, если сохранить число выполненных циклов при поиске на равно, то, умножив это число на 2 (SHL сдвиг влево на один бит), получим относительный адрес искомого значения цены. (Начальное значение счетчика циклов должно быть равно -1).
- Помните, что DB позволяет определять значения, не превышающие 256, а DW записывает байты в обратной последовательности. Команды CMP и CMPSW предполагают, что байты в сравниваемых словах имеют обратную последовательность.
- В случае, если таблица подвергается частым изменениям, или должна быть доступна нескольким программам, то запишите ее на диск. Для внесения изменений в таблицу можно разработать специальную программу модификации. Любые программы могут загружать таблицу с диска и при обновлениях таблицы сами программы не нуждаются в изменениях.
- Будьте особенно внимательны при кодировке сортирующих программ. Пользуйтесь трассировкой для тестирования, так как малейшая ошибка может привести к непредсказуемым результатам.

Лекция 7. Свойства операторов работы с экраном



Команда прерывания INT

Ранее мы имели дело с программами, в которых данные определялись в операндах команд (непосредственные данные) или инициализировались в конкретных полях программы. Число практических применений таких программ в действительности мало. Большинство программ требуют ввода данных с клавиатуры, диска или модема и обеспечивают вывод данных в удобном формате на экран, принтер или диск. Данные, предназначенные для вывода на экран и ввода с клавиатуры, имеют ASCII формат.

Для выполнения ввода и вывода используется команда INT (прерывание).

Существуют различные требования для указания системе какое действие (ввод или вывод) и на каком устройстве необходимо выполнить. Все необходимые экранные и клавиатурные операции можно выполнить используя команду INT 10H, которая передает управление непосредственно в BIOS. Для выполнения некоторых более сложных операций существует прерывание более высокого уровня INT 21H, которое сначала передает управление в DOS. Например, при вводе с клавиатуры может потребоваться подсчет введенных символов, проверку на максимальное число символов и проверку на символ Enter. Прерывание DOS INT 21H выполняет многие из этих дополнительных вычислений и затем автоматически передает управление в BIOS.

Команда INT прерывает обработку программы, передает управление в DOS или BIOS для определенного действия и затем возвращает управление в прерванную программу для продолжения обработки. Наибо-

лее часто прерывание используется для выполнения операций ввода или вывода. Для выхода из программы на обработку прерывания и для последующего возврата команда INT выполняет следующие действия:

- уменьшает указатель стека на 2 и заносит в вершину стека содержимое флагового регистра;
- очищает флаги TF и IF;
- уменьшает указатель стека на 2 и заносит содержимое регистра CS в стек;
- уменьшает указатель стека на 2 и заносит в стек значение командного указателя;
- обеспечивает выполнение необходимых действий;
- восстанавливает из стека значение регистра и возвращает управление в прерванную программу на команду, следующую после INT.

Этот процесс выполняется полностью автоматически. Необходимо лишь определить сегмент стека достаточно большим для записи в него значений регистров.



Установка курсора

Экран можно представить в виде двумерного пространства с адресуемыми позициями в любую из которых может быть установлен курсор. Обычный видеомонитор, например, имеет 25 строк (нумеруемых от 0 до 24) и 80 столбцов (нумеруемых от 0 до 79).

Команда INT 10H включает в себя установку курсора в любую позицию и очистку экрана. Ниже приведен пример установки курсора на 5-ю строку и 12-й столбец:

```
MOV AH,02 ;Запрос на установку курсора
MOV BH,00 ;Экран 0
MOV DH,05 ;Строка 05
MOV DL,12 ;Столбец 12
INT 10H ;Передача управления в BIOS
```


Значение 02 в регистре АН указывает команде INT 10H на выполнение операции установки курсора. Значение строки и столбца должны быть в регистре DX, а номер экрана (или страницы) в регистре ВН (обычно 0). Содержимое других регистров несущественно. Для установки строки и столбца можно также использовать одну команду MOV с непосредственным шест. значением:

```
MOV DX,050CH ;Строка 5, столбец 12
```



Очистка экрана

Запросы и команды остаются на экране пока не будут смещены в результате прокручивания («скроллинга») или переписаны на этом же месте другими запросами или командами.

Когда программа начинает свое выполнение, экран может быть очищен.

Очищаемая область экрана может начинаться в любой позиции и заканчиваться в любой другой позиции с большим номером.

Начальное значение строки и столбца заносится в регистр DX, значение 07 — в регистр ВН и 0600H в АХ. В следующем примере выполняется очистка всего экрана:

```
MOV AX,0600H ;AH 06 (прокрутка) ;AL 00 (весь экран)
MOV BH,07 ;Нормальный атрибут (черно/белый)
MOV CX,0000 ;Верхняя левая позиция
MOV DX,184FH ;Нижняя правая позиция
INT 10H ;Передача управления в BIOS
```

Значение 06 в регистре АН указывает команде INT 10H на выполнение операции очистки экрана.

Эта операция очищает экран пробелами. В случае, если вы по ошибке установили нижнюю правую позицию больше, чем шест. 184F, то очистка перейдет вновь к началу экрана и вторично заполнит некоторые позиции пробелами.



Использование символов возврата каретки, конца строки и табуляции для вывода на экран

Один из способов получения более эффективного выводе на экран — использование управляющих символов возврата каретки, перевода строки и табуляции:

Десятичные	**ASCII**	**Шестнадцатеричные**
CR	13	0DH
LF	10	0AH
TAB	09	09H

Эти символы при операциях ввода-вывода выполняют одинаковые действия как в базовой, так и в расширенной версиях DOS. Например:

```
MESSAGE DB 09,'PC Users Group Annual Report',13,10
MOV AH,40H ;Запрос на вывод
MOV BX,01 ;Номер файла
MOV CX,31 ;Длина текста
LEA DX,MESSAGE ;Адрес текста
INT 21H ;Вызов DOS
```

Использование директивы EQU для определения кодов делает программу более понятной:

```
CR EQU 13 ;или EQU 0DH LF
EQU 10 ;или EQU 0AH TAB
EQU 09 ;или EQU 09H
MESSAGE DB TAB, 'PC Users Group Annual' DB 'Report', CR, LF
```




Расширенные возможности экранных операций

Байт атрибутов

Байт атрибутов в текстовом (не графическом) режиме определяет характеристики каждого отображаемого символа. Байт-атрибут имеет следующие 8 бит:

```
                 Фон          Текст
Атрибут:         BL R G B     I R G B
Номер битов:     7 6 5 u      3 2 1 0
```

Буквы RGB представляют битовые позиции, управляющие красным (red), зеленым (green) и синим (blue) лучом в цветном мониторе. Бит 7 (BL) устанавливает мигание, а бит 3 (I) — уровень яркости.

Для модификации атрибутов можно комбинировать биты следующим образом:

Эффект выделения

	Фон	Текст
	RGB	RGB
Неотображаемый (черный по черному)	000	000
Подчеркивание (не для цвета)	000	001
Нормальный (белый по черному)	000	111
Инвертированный (черный по белому)	111	000

Цветные мониторы не обеспечивают подчеркивания; вместо этого установка бит подчеркивания выбирает синий цвет для текста и получается отображение синим по черному. Ниже приведены некоторые атрибуты, основанные на комбинации битов фона, текста, мигания и выделения яркостью:

Эффект выделения	Двоичный код	Шест. код
Неотображаемый (для паролей)	0000 0000	00
Белый по черному (нормальный)	0000 0111	07
Белый по черному (мигание)	1000 0111	87
Белый по черному (яркий)	0000 1111	0F
Черный по белому (инвертированный)	0111 0000	70
Черный по белому (инверт. мигающий)	1111 0000	F0

Эти атрибуты подходят для текстового режима, как для монохромных, так и для цветных дисплеев. Для генерации атрибута можно использовать команду INT 10H. При этом регистр BL должен содержать значение байта-атрибута, а регистр AH один из следующих кодов: 06 (прокрутка вверх), 07 (прокрутка вниз), 08 (ввод атрибута или символа), 09 (вывод атрибута или символа).

В случае, если программа установила некоторый атрибут, то он остается таким, пока программа его не изменит. В случае, если установить значение байта атрибута равным шест.00, то символ вообще не будет отображен.

Прерывание BIOS INT 10H

Прерывание INT 10H обеспечивает управление всем экраном. В регистре AH устанавливается код, определяющий функцию прерывания. Команда сохраняет содержимое регистров BX, CX, DX, SI и BP. Ниже описывается все возможные функции.

AH=00

Установка режима. Данная функция позволяет переключать цветной монитор в текстовый или графический режим. Установка режима для выполняемой в текущий момент программы осуществляется с помощью INT 10H.

AH=01

Установка размера курсора. Курсор не является символом из набора ASCII-кодов. Компьютер имеет собственное аппаратное обеспечение для управления видом курсора. Для этого имеется специальная обработка по INT прерыванию. Обычно символ курсора похож на символ подчеркивания.

AH=02

Установка позиции курсора. Эта функция устанавливает курсор в любую позицию на экране в соответствии с координатами строки и столбца.

AH=03

Чтение текущего положения курсора. Программа может определить положение курсора на экране (строку и столбец), а также размер курсора, следующим образом:

AH=04

Чтение положения светового пера. Данная функция используется в графическом режиме для определения положения светового пера.

AH=05

Выбор активной страницы.

AH=06

Прокрутка экрана вверх. Когда программа пытается выдать текст на строку ниже последней на экране, то происходит переход на верхнюю строку. Даже если с помощью прерывания будет специфицирован нулевой столбец, все равно предполагается новая строка, и нижние строки на экране будут испорчены. Для решения этой проблемы используется прокрутка экрана.

Ранее код 06 использовался для очистки экрана. В текстовом режиме установка в регистре AL значения 00 приводит к полной прокрутке вверх всего экрана, очищая его пробелами. Установка ненулевого значения в регистре AL определяет количество строк прокрутки экрана вверх. Верхние строки уходят с экрана, а чистые строки вводятся снизу. Следующие команды выполняют прокрутку всего экрана на одну строку:

```
MOV AX,0601H ;Прокрутить на одну строку вверх
MOV BH,07 ;Атрибут: нормальный, черно-белый
MOV CX,0000 ;Координаты от 00,00
MOV DX,184FH ; до 24,79 (полный экран)
INT 10H ;Вызвать BIOS
```

Для прокрутки любого количества строк необходимо установить соответствующее значение в регистре AL. Регистр BH содержит атрибут для нормального или инвертированного отображения, мигания, установки цвета и так далее. Значения в регистрах CX и DX позволяют прокручивать любую часть экрана. Ниже объясняется стандартный подход к прокрутке:

1. Определить в элементе ROW (строка) значение 0 для установки строки положения курсора.

2. Выдать текст и продвинуть курсор на следующую строку.

3. Проверить, находится ли курсор на последней строке (CMP ROW,22).

4. В случае, если да, то увеличить элемент ROW (INC ROW) и выйти.

5. В случае, если нет, то прокрутить экран на одну строку и, используя ROW переустановить курсор.

AH=07

Прокрутка экрана вниз. Для текстового режима прокрутка экрана вниз обозначает удаление нижних строк и вставка чистых строк сверху.

Регистр AH должен содержать 07, значения остальных регистров аналогичны функции 06 для прокрутки вверх.

AH=08

Чтение атрибута/символа в текущей позиции курсора. Для чтения символа и байта атрибута из дисплейного буфера, как в текстовом, так и в графическом режиме используются следующие команды:

```
MOV AH,08 ;Запрос на чтение атр./симв.
MOV BH,00 ;Страница 0 (для текстового реж.)
INT 10H ;Вызвать BIOS
```

Данная функция возвращает в регистре AL значение символа, а в AH — его атрибут. В графическом режиме функция возвращает шест.00 для не ASCII-кодов. Так как эта функция читает только один символ, то для символьной строки необходима организация цикла.

AH=09

Вывод атрибута/символа в текущую позицию курсора. Для вывода на экран символов в текстовом или графическом режиме с установкой мигания, инвертирования и так далее можно воспользоваться следующими командами:

```
MOV AH,09 ;Функция вывода
MOV AL,символ ;Выводимый символ
MOV BH,страница ;Номер страницы (текст.реж.)
MOV BL,атрибут ;Атрибут или цвет
MOV CX,повторение ;Число повторений символа
INT 10H ;Вызвать BIOS
```

В регистр AL должен быть помещен выводимый на экран символ. Значение в регистре CX определяет число повторений символа на экране. Вывод на экран последовательности различных символов требует организации цикла. Данная функция не перемещает курсор. В следующем примере на экран выводится пять мигающих «сердечек» в инвертированном виде:

```
MOV AH,09 ;Функция вывода
MOV AL,03H ;Черви (карточная масть)
MOV BH,00 ;Страница 0 (текст. режим)
MOV BL,0F0H ;Мигание, инверсия
MOV CX,05 ;Пять раз
INT 10H ;Вызвать BIOS
```

В текстовом (но не в графическом) режиме символы автоматически выводятся на экран и переходят с одной строки на другую. Для вывода на экран текста запроса или сообщения необходимо составить программу, которая устанавливает в регистре CX значение 01 и в цикле

загружает в регистр AL из памяти выводимые символы текста. Так как регистр CX в данном случае занят, то нельзя использовать команду LOOP. Кроме того, при выводе каждого символа необходимо дополнительно продвигать курсор в следующий столбец (функция 02).

В графическом режиме регистр BL используется для определения цвета графики. В случае, если бит 7 равен 0, то заданный цвет заменяет текущий цвет точки, если бит 7 равен 1, то происходит комбинация цветов с помощью команды XOR.

AH=0A

Вывод символа в текущую позицию курсора. Единственная разница между функциями 0A и 09 состоит в том, что функция 0A не устанавливает атрибут:

```
MOV AH,0AH ;Функция вывода
MOV AL,символ ;Выводимый символ
MOV BH,страница ;Номер страницы (для текста)
MOV CX,повторение ;Число повторений символа
INT 10H ;Вызвать BIOS
```

Для большинства применений команда прерывания DOS INT 21H более удобна.

AH=0E

Вывод в режиме телетайпа. Данная функция позволяет использовать монитор, как простой терминал.

Для выполнения этой функции необходимо установить в регистре AH шест. значение 0E, в регистр AL поместить выводимый символ, цвет текста (в графическом режиме) занести в регистр BL и номер страницы для текстового режима — в регистр BH. Звуковой сигнал (код 07H), возврат на одну позицию (08H), конец строки (0AH) и возврат каретки (0DH) действуют, как команды для форматизации экрана.

Данная функция автоматически продвигает курсор, переводит символы на следующую строку, выполняет прокрутку экрана и сохраняет текущие атрибуты экрана.

AH=0F

Получение текущего видео режима. Данная функция возвращает в регистре AL текущий видео режим, в регистре AH — число символов в строке (20, 40 или 80), в регистре BH — номер страницы.

AH=13

Вывод символьной строки (только для AT). Данная функция позволяет на компьютерах типа AT выводить на экран символьные строки с установкой атрибутов и перемещением курсора:

```
MOV AH,13H ;Функция вывода на экран
MOV AL,сервис ;0, 1, 2 или 3
MOV BH,страница ;
LEA BP,адрес ;Адрес строки в ES:BP
MOV CX,длина ;Длина строки
MOV DX,экран ;Координаты на экране
INT 10H ;Вызвать BIOS
```

Возможен следующий дополнительный сервис:

- 0 — использовать атрибут и не перемещать курсор;
- 1 — использовать атрибут и переместить курсор;
- 2 — вывести символ, затем атрибут и не перемещать курсор;
- 3 — вывести символ, затем атрибут и переместить курсор.



Расширенный ASCII код

ASCII-коды от 128 до 255 (шест. 80-FF) представляют собой ряд специальных символов полезных при формировании запросов, меню, специальных значков с экранными атрибутами. Например, используя следующие символы можно нарисовать прямоугольник:

`DA`

Верхний левый угол

`BF`

Верхний правый угол

`C0`

Нижний левый угол

`D9`

Нижний правый угол

`C4`

Горизонтальная линия

`B3`

Вертикальная линия

Следующие команды с помощью INT 10H выводят горизонтальную линию на 25 позиций в длину:

```
MOV AH,09 ;Функция вывода на экран
MOV AL,0C4H ;Горизонтальная линия
MOV BH,00 ;Страница 0
MOV BL,0FH ;Выделение яркостью
MOV CX,25 ;25 повторений
MOV 10H ;Вызвать BIOS
```

Напомним, что курсор не перемещается.

Вывод вертикальной линии включает цикл, в котором курсор перемещается вниз на одну строку и выводится символ шест. B3. Для штриховки может быть полезен символ с точками внутри:

`B0`

Одна четверть точек (светлая штриховка)

`B1`

Половина точек (средняя штриховка)

`B2`

Три четверти точек (темная штриховка)

Можно извлечь много полезных идей, изучая программное обеспечение с профессионально организованным выводом, или самому изобрести оригинальные идеи для отображения информации.



Другие операции ввода/вывода

Ниже перечислены другие функции DOS, которые могут оказаться полезными в работе. Код функции устанавливается в регистре AH и, затем, выдается команда INT 21H.

AH=01

Ввод с клавиатуры с эхо отображением. Данная функция возвращает значение в регистре AL. В случае, если содержимое AL не равно нулю, то оно представляет собой стандартный ASCII-символ, например, букву или цифру. Нулевое значение в регистре AL свидетельствует о том, что на клавиатуре была нажата специальная функциональная клавиша, например, Home, F1 или PgUp. Для определения скэн-кода клавиш, необходимо повторить вызов функции. Данная функция реагирует на запрос Ctrl/Break.

AH=02

Вывод символа. Для вывода символа на экран в текущую позицию курсора необходимо поместить код данного символа в регистр DL. Коды табуляции, возврата каретки и конца строки действуют обычным образом.

AH=07

Прямой ввод с клавиатуры без эхо отображения. Данная функция работает аналогично функции 01 с двумя отличиями: введенный символ не отображается на экране, то есть, нет эхо, и отсутствует реакция на запрос Ctrl/Break.

AH=08

Ввод с клавиатуры без эхо отображения. Данная функция действует аналогично функции 01 с одним отличием: введенный символ не отображается на экран, то есть, нет эхо.

AH=0B

Проверка состояния клавиатуры. Данная функция возвращает шест. FF в регистре AL, если ввод с клавиатуры возможен, в противном случае — 00. Это средство связано с функциями 01, 07 и 08, которые не ожидают ввода с клавиатуры.



Ввод с клавиатуры по команде BIOS INT 16H

Команда BIOS INT 16H выполняет специальную операцию, которая в соответствии с кодом в регистре AH обеспечивает следующие три функции ввода с клавиатуры.

AH=00

Чтение символа. Данная функция помещает в регистр AL очередной ASCII символ, введенный с клавиатуры, и устанавливает скэн-код в регистре AH. В случае, если на клавиатуре нажата одна из специальных клавишей, например, Home или F1, то в регистр AL заносится 00. Автоматическое эхо символа на экран по этой функции не происходит.

AH=01

Определение наличия введенного символа. Данная функция сбрасывает флаг нуля (ZF=0), если имеется символ для чтения с клавиатуры; очередной символ и скэн-код будут помещены в регистры AL и AH соответственно и данный элемент останется в буфере.

AH=02

Определение текущего состояния клавиатуры. Данная функция возвращает в регистре AL состояние клавиатуры из адреса памяти:

Бит

7

Состояние вставки активно (**Ins**)

6

Состояние фиксации верхнего регистра (**Caps Lock**) переключено

5

Состояние фиксации цифровой клавиатуры (**Num Lock**) переключено

4

Состояние фиксации прокрутки (**Scroll Lock**) переключено

3

Нажата комбинация клавишей **Alt/Shift**

2

Нажата комбинация клавишей **Ctrl/Shift**

1

Нажата левая клавиша **Shift**

0

Нажата правая клавиша **Shift**



Функциональные клавиши

Клавиатура располагает тремя основными типами клавишей:

1. Символьные (алфавитно-цифровые) клавиши: буквы от a до z, цифры от 0 до 9, символы %, $, # и так далее.

2. Функциональные клавиши: **Home**, **End**, Возврат на позицию, стрелки, **Enter**, **Del**, **Ins**, **PgUp**, **PgDn** и программно-функциональные клавиши.

3. Управляющие клавиши: **Alt**, **Ctrl** и **Shift**, которые работают совместно с другими клавишами.

Функциональная клавиша не вырабатывает какой-либо символ, но чаще формирует запрос на некоторые действия. Аппаратная реализация не требует от функциональных клавишей выполнения каких-либо специфических действий.

Задачей программиста является определить, например, что нажатие клавиши **Home** должно привести к установке курсора в верхний левый угол экрана, или нажатие клавиши **End** должно установить курсор в конец текста на экране.

Можно легко запрограммировать функциональные клавиши для выполнения самых различных действий.

Каждая клавиша имеет собственный скэн-код от 1 (**Esc**) до 83 (Del) или от шест.01 до шест.53. Посредством этих скэн-кодов программа может определить нажатие любой клавиши. Например, запрос на ввод одного символа с клавиатуры включает загрузку 00 в регистр AH и обращение к BIOS через INT 16H:

```
MOV AH,00 ;Функция ввода с клавиатуры
INT 16H ;Вызвать BIOS
```

Данная операция имеет два типа ответов в зависимости от того, нажата символьная клавиша или функциональная. Для символа (например, буква A) клавиатура посылает в компьютер два элемента информации:

1. ASCII-код символа A (шест.41) в регистре AL;

2. Скэн-код для клавиши A (шест.1E) в регистре AH.

В случае, если нажата функциональная клавиша (например, Ins) клавиатура также передает два элемента:

1. Нуль в регистре AL;

2. Скэн-код для клавиши Ins (шест.52) в регистре AH.

Таким образом, после выполнения команды INT 16H необходимо прежде проверить содержимое регистра AL. В случае, если AL содержит нуль, то была нажата функциональная клавиша, если не нуль, то получен код символьной клавиши.

Скэн-коды

Клавиатура имеет по две клавиши для таких символов как *, + и - . Нажатие «звездочки», например, устанавливает код символа шест.2A в регистре AL и один из двух скэн-кодов в регистре AH в зависимости от того, какая из клавишей была нажата: шест.09 для звездочки над цифрой 8 или шест.29 для звездочки на клавише PrtSc.

Функциональные клавиши	Скэн-коды
Alt/A – Alt/Z	1E – 2C
F1 – F10	3B – 44
Home	47
Стрелка вверх	48
PgUp	49
Стрелка влево	4B
Стрелка вправо	4D
End	4F
Стрелка вниз	50
PgDu	51
Ins	52
Del	53

Приведем пример программы для установки курсора в строку 0 и столбец 0 при нажатии клавиши Home (скэн-код 47):

```
MOV AH,00 ;Выполнить ввод с клавиатуры
INT 16H ; CMP AL,00 ;Функциональная клавиша?
JNE EXIT1 ; нет - выйти
CMP AH,47H ;Скэн-код для клавиши Home?
JNE EXIT2 ; нет - выйти
MOV AH,02 ;
MOV BH,00 ;Установить курсор
MOV DX,00 ; по координатам 0,0
INT 10H ;Вызвать BIOS
```

Функциональные клавиши F1—F10 генерируют скэн-коды от шест.3B до шест.44. Следующий пример выполняет проверку на функциональную клавишу F10:

```
CMP AH,44H ;Клавиша F10?
JE EXIT1 ; Да!
```

По адресу EXIT1 программа может выполнить любое необходимое действие.



Цвет и графика

Текстовой режим

Текстовой режим предназначен для обычных вычислений с выводом букв и цифр на экран. Данный режим одинаков для черно-белых (BW) и для цветных мониторов за исключением того, что цветные мониторы не поддерживают атрибут подчеркивания. Текстовой режим обеспечивает работу с полным набором ASCII кодов (256 символов), как для черно-белых (BW), так и для цветных мониторов. Каждый символ на экране может отображаться в одном из 16 цветов на одном из восьми цветов фона. Бордюр экрана может иметь также один из 16 цветов.

Цвета

Тремя основными цветами являются красный, зеленый и синий. Комбинируя основные цвета друг с другом, можно получить восемь цветов, включая черный и белый. Используя два уровня яркости для каждого цвета, получим всего 16 цветов:

	I	R	G	B
Черный	0	0	0	0
Серый	1	0	0	0
Синий	0	0	0	1
Ярко-синий	1	0	0	1
Зеленый	0	0	1	0
Ярко-зеленый	1	0	1	0
Голубой	0	0	1	1
Ярко-голубой	1	0	1	1
Красный	0	1	0	0
Ярко-красный	1	1	0	0
Сиреневый	0	1	0	1
Ярко-сиреневый	1	1	0	1
Коричневый	0	1	1	0
Желтый	1	1	1	0
Белый	0	1	1	0
Ярко-белый	1	1	1	1

Таким образом любые символы могут быть отображены на экране в одном из 16 цветов. Фон любого символа может иметь один из первых восьми цветов.

В случае, если фон и текст имеют один и тот же цвет, то текст получается невидимым.

Используя байт атрибута, можно получить также мигающие символы.

Байт-атрибут

Текстовой режим допускает использование байта атрибута. Цвет на экране сохраняется до тех пор, пока другая команда не изменит его.

Для установки цвета можно использовать в команде INT 10H функции AH=06, AH=07 и AH=09. Например, для вывода пяти мигающих звездочек светло-зеленым цветом на сиреневом фоне возможна следующая программа:

```
MOV AH,09 ;Функция вывода на экран
MOV AL,'*' ;Выводимый символ
MOV BH,00 ;Страница 0
MOV BL,0DAH ;Атрибут цвета
MOV CX,05 ;Число повторений
INT 10H ;Вызвать BIOS
```

Графический режим

Для генерации цветных изображений в графическом режиме используются минимальные точки растра — пикселы или пэлы (pixel). Цветной графический адаптер (CGA) имеет три степени разрешения:

1. Низкое разрешение.

2. Среднее разрешение.

3. Высокое разрешение.

Лекция 8. Требования языка



Комментарии в программах на Ассемблере

Использование комментариев в программе улучшает ее ясность, особенно там, где назначение набора команд непонятно. Комментарий всегда начинаются на любой строке исходного модуля с символа точка с запятой (;) и Ассемблер полагает в этом случае, что все символы, находящиеся справа от ; являются комментарием. Комментарий может содержать любые печатные символы, включая пробел. Комментарий может занимать всю строку или следовать за командой на той же строке, как это показано в двух следующих примерах:

```
;Эта строка полностью является комментарием
ADD AX,BX ;Комментарий на одной строке с командой
```

Комментарии появляются только в листингах ассемблирования исходного модуля и не приводят к генерации машинных кодов, поэтому можно включать любое количество комментариев, не оказывая влияния на эффективность выполнения программы.



Формат кодирования

Основной формат кодирования команд Ассемблера имеет следующий вид:

```
[метка] команда [операнд(ы)]
```


Метка (если имеется), команда и операнд (если имеется) разделяются по крайней мере одним пробелом или символом табуляции. Максимальная длина строки — 132 символа, однако, большинство предпочитают работать со строками в 80 символов (соответственно ширине экрана). Примеры кодирования:

```
COUNT DB 1 ;Имя, команда, один операнд
MOV AX,0 ;Команда, два операнда
```

Метки

Метка в языке Ассемблера может содержать следующие символы:

Буквы: от A до Z и от a до z

Цифры: от 0 до 9

Спецсимволы: знак вопроса (?) точка (.) (только первый символ) знак «коммерческое эт» (@) подчеркивание (-) доллар ($)

Первым символом в метке должна быть буква или спецсимвол. Ассемблер не делает различия между заглавными и строчными буквами. Максимальная длина метки — 31 символ.

Примеры меток:

```
COUNT
PAGE25
$E10
```

Рекомендуется использовать описательные и смысловые метки. Имена регистров, например, AX, DI или AL являются зарезервированными и используются только для указания соответствующих регистров. Например, в команде

```
ADD AX,BX
```

Ассемблер «знает», что AX и BX относится к регистрам. Однако, в команде

```
MOV REGSAVE,AX
```

Ассемблер воспримет имя REGSAVE только в том случае, если оно будет определено в сегменте данных.

Команда

Мнемоническая команда указывает Ассемблеру какое действие должен выполнить данный оператор. В сегменте данных команда (или директива) определяет поле, рабочую область или константу.

В сегменте кода команда определяет действие, например, пересылка (MOV) или сложение (ADD).

Операнд

В случае, если команда специфицирует выполняемое действие, то операнд определяет **начальное значение данных** или **элементы**, над которыми выполняется действие по команде. В следующем примере байт COUNTER определен в сегменте данных и имеет нулевое значение:

```
COUNTER DB 0 ;Определить байт (DB) с нулевым значением
```

Команда может иметь один или два операнда, или вообще быть без операндов. Рассмотрим следующие три примера:

Нет операндов

```
RET ;Вернуться
```

Один операнд

```
INC CX ;Увеличить CX
```

Два операнда

```
ADD AX,12 ;Прибавить 12 к AX
```

Метка, команда и операнд не обязательно должны начинаться с какой-либо определенной позиции в строке. Однако, рекомендуется записывать их в колонку для большей удобочитаемости программы. Для этого, например, редактор DOS EDLIN обеспечивает табуляцию через каждые восемь позиций.



Директивы

Ассемблер имеет ряд операторов, которые позволяют управлять процессом ассемблирования и формирования листинга.

Эти операторы называются псевдокомандами или директивами.

Они действуют только в процессе ассемблирования программы и не генерируют машинных кодов.

Директивы управления листингом: PAGE и TITLE

Ассемблер содержит ряд директив, управляющих форматом печати (или листинга). Обе директивы PAGE и TITLE можно использовать в любой программе.

Директива PAGE

В начале программы можно указать количество строк, распечатываемых на одной странице, и максимальное количество символов на одной строке. Для этой цели служит директива PAGE. Следующей директивой устанавливается 60 строк на страницу и 132 символа в строке:

```
PAGE 60,132
```

Количество строк на странице может быть в пределах от 10 до 255, а символов в строке — от 60 до 132. По умолчанию в Ассемблере установлено:

```
PAGE 66,80
```

Предположим, что счетчик строк установлен на 60. В этом случае Ассемблер, распечатав 60 строк, выполняет прогон листа на начало следующей страницы и увеличивает номер страницы на единицу. Кроме того можно заставить Ассемблер сделать прогон листа на конкретной строке, например, в конце сегмента. Для этого необходимо записать директиву PAGE без операндов. Ассемблер автоматически делает прогон листа при обработке директивы PAGE.

Директива TITLE

Для того, чтобы вверху каждой страницы листинга печатался заголовок (титул) программы, используется директива TITLE в следующем формате:

```
TITLE текст
```

Рекомендуется в качестве текста использовать имя программы, под которым она находится в каталоге на диске. Например, если программа называется ASMSORT, то можно использовать это имя и описательный комментарий общей длиной до 60 символов:

```
TITLE ASMSORT - Ассемблерная программа сортировки имен
```

В Ассемблере также имеется директива подзаголовка SUBTTL, которая может оказаться полезной для очень больших программ, содержащих много подпрограмм.

Директива SEGMENT

Любые ассемблерные программы содержат по крайней мере один сегмент — сегмент кода. В некоторых программах используется сегмент

для стековой памяти и сегмент данных для определения данных. Ассемблерная директива для описания сегмента SEGMENT имеет следующий формат:

```
Имя Директива Операнд имя SEGMENT [параметры]
...
...
имя ENDS
```

Имя сегмента должно обязательно присутствовать, быть уникальным и соответствовать соглашениям для имен в Ассемблере. Директива ENDS обозначает конец сегмента. Обе директивы SEGMENT и ENDS должны иметь одинаковые имена. Директива SEGMENT может содержать три типа параметров, определяющих выравнивание, объединение и класс.

Выравнивание

Данный параметр определяет границу начала сегмента. Обычным значением является PARA, по которому сегмент устанавливается на границу параграфа. В этом случае начальный адрес делится на 16 без остатка, то есть, имеет шест. адрес nnn0. В случае отсутствия этого операнда Ассемблер принимает по умолчанию PARA.

Объединение

Этот элемент определяет объединяется ли данный сегмент с другими сегментами в процессе компоновки после ассемблирования. Возможны следующие типы объединений: STACK, COMMON, PUBLIC, AT выражение и MEMORY.

Сегмент стека определяется следующим образом:

```
имя SEGMENT PARA STACK
```

Когда отдельно ассемблированные программы должны объединяться компоновщиком, то можно использовать типы: PUBLIC, COMMON и MEMORY. В случае, если программа не должна объединяться с другими программами, то данная опция может быть опущена.

Класс

Данный элемент, заключенный в апострофы, используется для группирования относительных сегментов при компоновке:

```
имя SEGMENT PARA STACK 'Stack'
```

Директива PROC

Сегмент кода содержит выполняемые команды программы. Кроме того этот сегмент также включает в себя одну или несколько процедур, определенных директивой PROC. Сегмент, содержащий только одну процедуру имеет следующий вид:


```
имя-сегмента SEGMENT PARA
имя-процедуры PROC
FAR
Сегмент кода с одной процедурой
имя-процедуры ENDP
имя-сегмента ENDS
RET
```

Имя процедуры должно обязательно присутствовать, быть уникальным и удовлетворять соглашениям по именам в Ассемблере. Операнд FAR указывает загрузчику DOS, что начало данной процедуры является точкой входа для выполнения программы.

Директива ENDP определяет конец процедуры и имеет имя, аналогичное имени в директиве PROC. Команда RET завершает выполнение программы и в данном случае возвращает управление в DOS.

Сегмент может содержать несколько процедур.

Директива ASSUME

Процессор использует регистр SS для адресации стека, регистр DS для адресации сегмента данных и регистр CS для адресации сегмента кода.

Ассемблеру необходимо сообщить назначение каждого сегмента. Для этой цели служит директива ASSUME, кодируемая в сегменте кода следующим образом:

```
Директива Операнд ASSUME SS:имя_стека,DS:имя_с_данных,
CS:имя_с_кода
```

Например, SS:имя_стека указывает, что Ассемблер должен ассоциировать имя сегмента стека с регистром SS. Операнды могут записываться в любой последовательности. Регистр ES также может присутствовать в числе операндов. В случае, если программа не использует регистр ES, то его можно опустить или указать ES:NOTHING.

Директива END

Директива ENDS завершает сегмент, а директива ENDP завершает процедуру. Директива END в свою очередь полностью завершает всю программу:

```
Директива Операнд END [имя_процедуры]
```

Операнд может быть опущен, если программа не предназначена для выполнения, например, если ассемблируются только определения данных, или эта программа должна быть скомпонована с другим (главным) модулем. Для обычной программы с одним модулем операнд содержит имя, указанное в директиве PROC, которое было обозначено как FAR.



Память и регистры

Рассмотрим особенности использования в командах имен, имен в квадратных скобках и чисел. В следующих примерах положим, что WORDA определяет слово в памяти:

```
MOV AX,BX ;Переслать содержимое BX в регистр AX
MOV AX,WORDA ;Переслать содержимое WORDA в регистр AX
MOV AX,[BX] ;Переслать содержимое памяти по адресу ; в регистре
BX в регистр AX
MOV AX,25 ;Переслать значение 25 в регистр AX MOV AX,[25] ;Пере-
слать содержимое по смещению 25
```

Новым здесь является использование квадратных скобок, что потребуется далее.



Инициализация программы

Существует два основных типа загрузочных программ: EXE и COM.

Рассмотрим требования к EXE-программам. DOS имеет четыре требования для инициализации ассемблерной EXE-программы:

1) указать Ассемблеру, какие сегментные регистры должны соответствовать сегментам;

2) сохранить в стеке адрес, находящийся в регистре DS, когда программа начнет выполнение;

3) записать в стек нулевой адрес;

4) загрузить в регистр DS адрес сегмента данных.

Выход из программы и возврат в DOS сводится к использованию команды RET.

Ассоциируя сегменты с сегментными регистрами, Ассемблер сможет определить смещения к отдельным областям в каждом сегменте. Например, каждая команда в сегменте кодов имеет определенную длину: первая команда имеет смещение 0, и если это двухбайтовая команда, то вторая команда будет иметь смещение 2 и так далее.

Загрузочному модулю в памяти непосредственно предшествует 256-байтовая (шест.100) область, называемая префиксом программного сегмента PSP. Программа загрузчика использует регистр DS для установки адреса начальной точки PSP. Пользовательская программа должна сохранить этот адрес, поместив его в стек. Позже, команда RET использует этот адрес для возврата в DOS.

В системе требуется, чтобы следующее значение в стеке являлось нулевым адресом (точнее, смещением). Для этого команда SUB очищает регистр AX, вычитая его из этого же регистра AX, а команда PUSH заносит это значение в стек.

Загрузчик DOS устанавливает правильные адреса стека в регистре SS и сегмента кодов в регистре CS. Поскольку программа загрузчика использует регистр DS для других целей, необходимо инициализировать регистр DS двумя командами MOV.

Команда RET обеспечивает выход из пользовательской программы и возврат в DOS, используя для этого адрес, записанный в стек в начале программы командой PUSH DS. Другим обычно используемым выходом является команда INT 20H.

Системный загрузчик при загрузке программы с диска в память для выполнения устанавливает действительные адреса в регистрах SS и CS. Программа не имеет сегмента данных, так как в ней нет определения данных и, соответственно, в ASSUME нет необходимости ассигновать регистр DS.

Команды PUSH, SUB и PUSH выполняют стандартные действия для инициализации стека текущим адресом в регистре DS и нулевым адресом. Поскольку, обычно, программа выполняется из DOS, то эти команды обеспечивают возврат в DOS после завершения программы.

(Можно также выполнить программу из отладчика, хотя это особый случай).

Важно:

- Не забывайте ставить символ «точка с запятой» перед комментариями.
- Завершайте каждый сегмент директивой ENDS, каждую процедуру — директивой ENDP, а программу — директивой END.
- В директиве ASSUME устанавливайте соответствия между сегментными регистрами и именами сегментов.
- Для EXE-программ обеспечивайте не менее 32 слов для стека, соблюдайте соглашения по инициализации стека командами PUSH, SUB и PUSH и заносите в регистр DS адрес сегмента данных.

Лекция 9. Ввод и выполнение программ



Ввод программы

Исходный текст программы, предназначенный для ввода с помощью текстового редактора можно ввести при помощи DOS EDLIN или другого текстового редактора. Для ввода исходной программы наберите команду:

```
EDLIN имя программы.ASM [Enter]
```

В результате DOS загрузит EDLIN в памяти и появится сообщение «New file» и приглашение «*-». Введите команду I для ввода строк, и затем наберите каждую ассемблерную команду.

Хотя число пробелов в тексте для Ассемблера не существенно, старайтесь записывать метки, команды, операнды и комментарии, выровненными в колонки, программа будет более удобочитаемая. Для этого в EDLIN используется табуляция через каждые восемь позиций.

После ввода программы убедитесь в ее правильности. Затем наберите E (и Enter) для завершения EDLIN. Можно проверить наличие программы в каталоге на диске, введите:

```
DIR (для всех файлов)
```

или

```
DIR имя программы.ASM (для одного файла)
```

В случае, если предполагается ввод исходного текста большего объема, то лучшим применением будет полноэкранный редактор. Для получения распечатки программы включите принтер и установите в него бумагу. Вызовите программу PRINT. DOS загрузит программу в память и распечатает текст на принтере:

```
PRINT имя программы.ASM [Enter]
```

Программа еще не может быть выполнена — прежде необходимо провести ее ассемблирование и компоновку.



Подготовка программы для выполнения

После ввода на диск исходной программы необходимо проделать два основных шага, прежде чем программу можно будет выполнить. Сначала необходимо ассемблировать программу, а затем выполнить компоновку. Программисты на языке бейсик могут выполнить программу сразу после ввода исходного текста, в то время как для Ассемблера и компилярных языков нужны шаги трансляции и компоновки.

Шаг ассемблирования включает в себя трансляцию исходного кода в машинный объектный код и генерацию OBJ-модуля. OBJ-модуль уже более приближен к исполнительной форме, но еще не готов к выполнению.

Шаг компоновки включает преобразование OBJ-модуля в EXE (исполнимый) модуль, содержащий машинный код. Программа LINK, находящаяся на диске DOS, выполняет следующее:

1. Завершает формирование в OBJ-модуле адресов, которые остались неопределенными после ассемблирования. Во многих следующих программах такие адреса Ассемблер отмечает как R.

2. Компонует, если необходимо, более одного отдельно ассемблированного модуля в одну загрузочную (выполнимую) программу; возможно две или более ассемблерных программ или ассемблерную программу с программами, написанными на языках высокого уровня, таких как Паскаль или Бейсик.

3. Инициализирует EXE-модуль командами загрузки для выполнения.

После компоновки OBJ-модуля (одного или более) в EXE-модуль, можно выполнить EXE-модуль любое число раз. Но, если необходимо внести некоторые изменения в EXE-модуль, следует скорректировать исходную программу, ассемблировать ее в другой OBJ-модуль и выполнить компоновку OBJ-модуля в новый EXE-модуль. Даже, если эти шаги пока остаются непонятными, вы обнаружите, что, получив немного навыка, весь процесс подготовки EXE-модуля будет доведен до автоматизма. Заметьте: определенные типы EXE-программ можно преобразовать в очень эффективные COM-программы.



Ассемблирование программы

Для того, чтобы выполнить исходную ассемблерную программу, необходимо прежде провести ее ассемблирование и затем компоновку. На дискете с ассемблерным пакетом имеются две версии ассемблера. ASM.EXE — сокращенная версия с отсутствием некоторых незначительных возможностей и MASM.EXE — полная версия. В случае, если размеры памяти позволяют, то используйте версию MASM.

Простейший вариант вызова программы ассемблирования — это ввод команды MASM (или ASM), что приведет к загрузке программы Ассемблера с диска в память. На экране появится:

```
source filename [.ASM]:
object filename [filename.OBJ]:
source listing [NUL.LST]:
cross-reference [NUL.CRF]:
```

Курсор при этом расположится в конце первой строки, где необходимо указать имя файла. Не следует набирать тип файла ASM, так как Ассемблер подразумевает это.

Во-втором запросе предполагается аналогичное имя файла (но можно его заменить).

Третий запрос предполагает, что листинг ассемблирования программы не требуется.

Последний запрос предполагает, что листинг перекрестных ссылок не требуется.

В случае, если вы хотите оставить значения по умолчанию, то в трех последних запросах просто нажмите **Enter**.

Всегда необходимо вводить имя исходного файла и, обычно, запрашивать OBJ-файл — это требуется для компоновки программы в загрузочный файл.

Возможно потребуется указание LST-файла, особенно, если необходимо проверить сгенерированный машинный код. CRF-файл полезен для очень больших программ, где необходимо видеть, какие команды ссылаются на какие поля данных. Кроме того, Ассемблер генерирует в LST-файле номера строк, которые используются в CRF-файле.

Ассемблер преобразует исходные команды в машинный код и выдает на экран сообщения о возможных ошибках. Типичными ошибками являются нарушения ассемблерных соглашений по именам, неправильное написание команд (например, MOVE вместо MOV), а также наличие в операндах неопределенных имен. Программа ASM выдает только коды ошибок, которые объяснены в руководстве по Ассемблеру, в то время как программа MASM выдает и коды ошибок, и пояснения к ним. Всего имеется около 100 сообщений об ошибках.

Ассемблер делает попытки скорректировать некоторые ошибки, но в любом случае следует перезагрузить текстовый редактор, исправить исходную программу и повторить ассемблирование.

Листинг содержит не только исходный текст, но также слева транслированный машинный код в шестнадцатеричном формате. В самой левой колонке находится шест.адреса команд и данных.

За листингом ассемблирования программы следует таблица идентификаторов. Первая часть таблицы содержит определенные в программе сегменты и группы вместе с их размером в байтах, выравниванием и классом.

Вторая часть содержит идентификаторы — имена полей данных в сегменте данных и метки, назначенные командам в сегменте кодов. Для того, чтобы Ассемблер не создавал эту таблицу, следует указать параметр **/N** вслед за командой MASM, то есть:

```
MASM/N
```

Двухпроходный Ассемблер

В процессе трансляции исходной программы Ассемблер делает два просмотра исходного текста, или два прохода. Одной из основных причин этого являются ссылки вперед, что происходит в том случае, когда в некоторой команде кодируется метка, значение которой еще не определено Ассемблером.

В первом проходе Ассемблер просматривает всю исходную программу и строит таблицу идентификаторов, используемых в программе, то есть, имен полей данных и меток программы и их относительных адресов в программе. В первом проходе подчитывается объем объектного кода, но сам объектный код не генерируется.

Во втором проходе Ассемблер использует таблицу идентификаторов, построенную в первом проходе. Так как теперь уже известны длины и относительные адреса всех полей данных и команд, то Ассемблер может сгенерировать объектный код для каждой команды. Ассемблер создает, если требуется, файлы: OBJ, LST и CRF.



Компоновка программы

В случае, если в результате ассемблирования не обнаружено ошибок, то следующий шаг — компоновка объектного модуля. Файл имяпрограммы.OBJ содержит только машинный код в шестнадцатеричной форме. Так как программа может загружаться почти в любое место памяти для выполнения, то Ассемблер может не определить все машинные адреса. Кроме того, могут использоваться другие (под) программы для объединения с основной. Назначением программы LINK является завершение определения адресных ссылок и объединение (если требуется) нескольких программ.

Для компоновки ассемблированной программы введите команду **LINK** и нажмите клавишу **Enter**. После загрузки в память, компоновщик выдает несколько запросов (аналогично MASM), на которые необходимо ответить:

`Object Modules [.OBJ]: имя программы`

Компонует имя программы.OBJ

`Run file [имя программы.EXE]:`

Создает имя программы.EXE

`List file [NUL.MAP]: CON`

Создает имя программы.MAP

`Libraries [.LIB]: [Enter]`

По умолчанию

Первый запрос — запрос имен объектных модулей для компоновки, тип OBJ можно опустить.

Второй запрос — запрос имени исполнимого модуля (файла), (по умолчанию имя программы.EXE). Практика сохранения одного имени (при разных типах) файла упрощает работу с программами.

Третий запрос предполагает, что LINK выбирает значение по умолчанию — NUL.MAP (то есть, MAP отсутствует). MAP-файл содержит таблицу имен и размеров сегментов и ошибки, которые обнаружит LINK. Типичной ошибкой является неправильное определение сегмента стека. Ответ CON предполагает, что таблица будет выведена на экран, вместо записи ее на диск. Это позволяет сэкономить место в дисковой памяти и сразу просмотреть таблицу непосредственно на экране.

Для ответа на четвертый запрос — нажмите **Enter**, что укажет компоновщику LINK принять остальные параметры по умолчанию.

На данном этапе единственной возможной ошибкой может быть указание неправильных имен файлов. Исправить это можно только перезапуском программы LINK.



Выполнение программы

После ассемблирования и компоновки программы можно (наконец-то!) выполнить ее. В случае, если EXE-файл находится на дисководе C, то выполнить ее можно командой:

```
C:имя программы.EXE или C:имя программы
```

DOS предполагает, что файл имеет тип EXE (или COM), и загружает файл для выполнения. Но так как наша программа не вырабатывает видимых результатов, выполним ее трассировкой под отладчиком DEBUG. Введите:

```
DEBUG C:имя программы.EXE
```

В результате DOS загрузит программу DEBUG, который, в свою очередь, загрузит требуемый EXE-модуль. После этого отладчик выдаст дефис (-) в качестве приглашения. Для просмотра сегмента стека введите

```
D SS:0
```

Эту область легко узнать по 12-кратному дублированию константы STACKSEG. Для просмотра сегмента кода введите

```
D CS:0
```

Введите R для просмотра содержимого регистров и выполните программу с помощью команды T (трассировка). Обратите внимание на

воздействие двух команд PUSH на стек — в вершине стека теперь находится содержимое регистра DS и нулевой адрес.

В процессе пошагового выполнения программы обратите внимание на содержимое регистров. Когда вы дойдете до команды RET, можно ввести Q (Quit — выход) для завершения работы отладчика.

Используя команду dir, можно проверить наличие ваших файлов на диске:

```
DIR C:имя программы.*
```

В результате на экране появится следующие имена файлов: имя программы.BAK (если для корректировки имя программы.ASM использовался редактор EDLIN), имя программы.ASM, имя программы.OBJ, имя программы.LST, имя программы.EXE и имя программы.CRF.

Последовательность этих файлов может быть иной в зависимости от того, что уже находится на диске.

Очевидно, что разработка ряда программ приведет к занятию дискового пространства. Для проверки оставшегося свободного места на диске полезно использовать команду **DOS CHKDSK**. Для удаления OBJ-, CRF-, BAK- и LST-файлов с диска следует использовать команду **ERASE** (или DEL):

```
ERASE C:имя программы.OBJ, ...
```

Следует оставить (сохранить) ASM-файл для последующих изменений и EXE-файл для выполнения.



Файл перекрестных ссылок

В процессе трансляции Ассемблер создает таблицу идентификаторов (CRF), которая может быть представлена в виде листинга перекрестных ссылок на метки, идентификаторы и переменные в программе. Для получения данного фала, необходимо на четвертый запрос Ассемблера, ответить C:, полагая, что файл должен быть создан на диске C:

```
cross-reference [NUL.CRF]:C: [Enter]
```

Далее необходимо преобразовать полученный CRF-файл в отсортированную таблицу перекрестных ссылок. Для этого на ассемблерном диске имеется соответствующая программа.

После успешного ассемблирования введите команду CREF. На экране появится два запроса:

```
Cref filename [.CRF]: List filename [cross-ref.REF]:
```

На первый запрос введите имя CRF-файла, то есть, С:имя программы. На второй запрос можно ввести только номер дисковода и получить имя по умолчанию.

Такой выбор приведет к записи CRF в файл перекрестных ссылок по имени имя программы.REF на дисководе С.

Для распечатки файла перекрестных ссылок используйте команду DOS PRINT.

Важно:

- Ассемблер преобразует исходную программу в OBJ-файл, а компоновщик — OBJ-файл в загрузочный EXE-файл.
- Внимательно проверяйте запросы и ответы на них для программ (M)ASM, LINK и CREF прежде чем нажать клавишу Enter. Будьте особенно внимательны при указании дисковода.
- Программа CREF создает распечатку перекрестных ссылок.
- Удаляйте ненужные файлы с вашего диска. Регулярно пользуйтесь программой CHKDSK для проверки свободного места на диске. Кроме того периодически создавайте резервные копии вашей программы, храните резервную дискету и копируйте ее заново для последующего программирования.

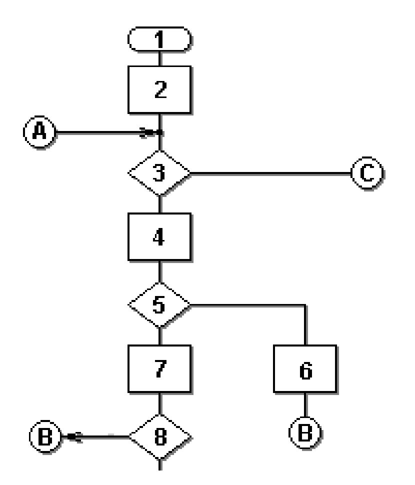
Лекция 10. Алгоритмы работы Ассемблеров

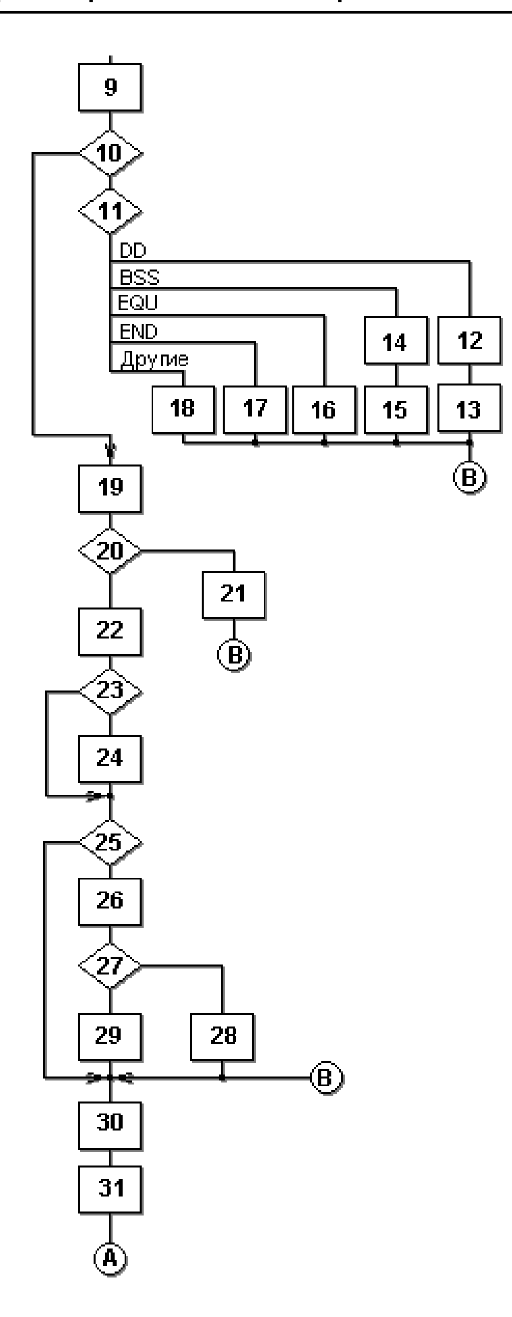


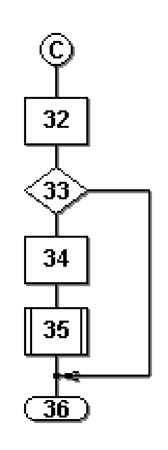
Двухпроходный Ассемблер — первый проход

Ассемблер просматривает исходный программный модуль один или несколько раз. Наиболее распространенными являются двухпроходные Ассемблеры, выполняющие два просмотра исходного модуля. На первом проходе Ассемблер формирует таблицу символов модуля, а на втором — генерирует код программы

Алгоритм работы 1-го прохода двухпроходного Ассемблера показан на рисунке.







- **Блок1:** Начало 1-го прохода ассемблирования.
- **Блок2:** Начальные установки:
  - установка в 0 счетчика адреса PC;
  - создание пустой таблицы символов;
  - создание пустой таблицы литералов;
  - открытие файла исходного модуля;
  - установка в FASLE признака окончания.
- **Блок3:** Признак окончания TRUE?
- **Блок4:** Считывание следующей строки исходного модуля. Добавка к счетчику адреса устанавливается равной 0.
- **Блок5:** При считывании был обнаружен конец файла?
- **Блок6:** Если конец файла обнаружен до того, как обработана директива END, — ошибка (преждевременный конец файла), при этом также устанавливается признак окончания обработки.
- **Блок7:** Лексический разбор оператора программы. При этом:
  - выделяется метка/имя, если она есть;
  - выделяется мнемоника операции;
  - выделяется поле операндов;
  - удаляются комментарии в операторе;

- распознается строка, содержащая только комментарий.

- **Блок8:** Строка содержит только комментарий? В этом случае обработка оператора не производится.
- **Блок9:** Мнемоника операции ищется в таблице директив.
- **Блок10:** Завершился ли поиск в таблице директив успешно?
- **Блок11:** Если мнемоника была найдена в таблице директив, происходит ветвление, в зависимости от того, какая директива была опознана.
- **Блок12:** Обработка директив типа DD (определения данных) включает в себя:
  - выделение элементов списка операндов (одной директивой DD может определяться несколько объектов данных);
  - определение типа и, следовательно, размера объекта данных, заданного операндом;
  - обработка для каждого операнда возможного коэффициента повторения.
- **Блок13:** Добавка к счетчику адреса устанавливается равной суммарному размеру объектов данных, определяемых директивой.
- **Блок14:** Обработка директив типа BSS подобна обработке директив типа DD.
- **Блок15:** Добавка к счетчику адреса устанавливается равной суммарному объему памяти, резервируемому директивой.
- **Блок16:** Обработка директивы END состоит в установке в TRUE признака окончания обработки.
- **Блок17:** Обработка директивы включает в себя вычисление значения имени и занесение его в таблицу символов.
- **Блок18:** Обработка прочих директив ведется по индивидуальным для каждой директивы алгоритмам. Существенно, что никакие директивы, кроме DD и BSS, не изменяют нулевого значения добавки к счетчику адреса.
- **Блок19:** Если мнемоника операции не найдена в таблице директив, она ищется в таблице команд.

- **Блок20:** Завершился ли поиск в таблице команд успешно?
- **Блок21:** Если мнемоника не была найдена в таблице команд, — ошибка (неправильная мнемоника).
- **Блок22:** Если мнемоника найдена в таблице команд — определение длины команды, она же будет добавкой к счетчику адреса.
- **Блок23:** Есть ли в операторе литерал?
- **Блок24:** Занесение литерала в таблицу литералов (если его еще нет в таблице).
- **Блок25:** Была ли в операторе метка?
- **Блок26:** Поиск имени в таблице символов.
- **Блок27:** Имя в таблице символов найдено?
- **Блок28:** Если имя найдено в таблице символов — ошибка (повторяющееся имя).Если имя не найдено в таблице символов — занесение имени в таблицу символов.
- **Блок29:** Формирование и печать строки листинга.
- **Блок30:** Модификация счетчика адреса вычисленной добавкой к счетчику
- **Блок31:** Печать строки листинга и переход к чтению следующего оператора.
- **Блок32:** При окончании обработки — закрытие файла исходного модуля.
- **Блок33:** Были ли ошибки на 1-м проходе ассемблирования?
- **Блок34:** Формирование литерального пула.
- **Блок35:** Выполнение 2-го прохода ассемблирования.
- **Блок36:** Конец работы Ассемблера.

Определение длины команды

Эта задача может решаться существенно разным образом для разных языков. В языках некоторых Ассемблеров мнемоника команды однозначно определяет ее формат и длину (S/390, все RISC-процессоры). В этом случае длина команды просто выбирается из таблицы команд. В других языках длина и формат зависит от того, с какими операндами употреблена команда (Intel). В этом случае длина вычисляется по некоторо-

му специфическому алгоритму, в который входит выделение отдельных операндов и определение их типов. В последнем случае должна производиться также необходимая проверка правильности кодирования операндов (количество операндов, допустимость типов).

Обнаружение литералов

Требует, как минимум, выделения операндов команды.

Листинг

Строка листинга печатается в конце каждой итерации обработки команды. Строка листинга 1-го прохода содержит: номер оператора, значение счетчика адреса (только для команд и директив, приводящих к выделению памяти), текст оператора. Листинг 1-го прохода не является окончательным, фактически он используется только для поиска ошибок, поэтому печатать его необязательно. Режим печати листинга 1-го прохода может определяться параметром Ассемблера или специальной директивой. После листинга программы может (опционно) печататься таблица символов.

Ошибки

На первом проходе выявляются не все ошибки, а только те, которые связаны с выполнением задачи 1-го прохода. Сообщение об ошибке включает в себя: код ошибки, диагностический текст, номер и текст оператора программы, в котором обнаружена ошибка.

Некоторые структуры данных 1-го прохода

Таблица команд содержит одну строку для каждой мнемоники машинной команды. Ниже приведен пример структуры такой таблицы для простого случая, когда мнемоника однозначно определяет формат и длину команды. Обработка команд происходит по всего нескольким алгоритмам, зависящим от формата команды, поэтому в данном случае все параметры обработки могут быть представлены в виде данных, содержащихся в таблице.

Таблица директив содержит одну строку для каждой директивы Обработка каждой директивы происходит по индивидуальному алгоритму, поэтому параметры обработки нельзя представить в виде данных единого для всех директив формата. Для каждой директивы в таблице хранится только идентификация (имя или адрес, или номер) процедуры Ассемблера, выполняющей эту обработку. Некоторые директивы обрабатываются только на 1-м проходе, некоторые — только на 2-м, для некоторых обработка распределяется между двумя проходами.

Таблица символов является основным результатом 1-го прохода Ассемблера. Каждое имя, определенное в программе, должно быть записано в таблице символов. Для каждого имени в таблице хранится его значение, размер объекта, связанного с этим именем и признак перемещаемости/неперемещаемости. Значением имени является число, в большинстве случаев интерпретируемое как адрес, поэтому разрядность значения равна разрядности адреса.

Перемещаемость рассматривается в разделе, посвященном Загрузчикам, здесь укажем только, что значение перемещаемого имени должно изменяться при загрузке программы в память. Имена, относящиеся к командам или к памяти, выделяемой директивами DD, BSS, как правило, являются перемещаемыми (относительными), имена, значения которых определяются директивой EQU, являются неперемещаемыми (абсолютными).

Символьное имя	Значение	Длина (байт)	Признак перемешаемости

Таблица литералов содержит запись для каждого употребленного в модуле литерала. Для каждого литерала в таблице содержится его символьное обозначение, длина и ссылка на значение. Литерал представляет собой константу, записанную в памяти. Обращение к литералам производится так же, как и к именам ячеек программы. По мере обнаружения в программе литералов Ассемблер заносит их данные в так называемый литеральный пул. Значение, записываемое в таблицу литералов является смещением литерала в литеральном пуле. После окончания 1-го прохода Ассемблер размещает литеральный пул в конце программы (то есть, назначает ему адрес, соответствующий последнему значению счетчик адреса) и корректирует значения в таблице литералов, заменяя их смещениями относительно начала программы. После выполнения этой корректировки таблица литералов может быть совмещена с таблицей символов.





Структура таблиц Ассемблера

Структура таблиц Ассемблера выбирается таким образом, чтобы обеспечить максимальную скорость поиска в них.

Таблицы команд и директив являются постоянной базой данных. Они заполняются один раз — при разработке Ассемблера, а затем остаются неизменными.

Эти таблицы целесообразно строить как таблицы прямого доступа с функцией хеширования, осуществляющей преобразование мнемоники в адрес записи в таблице.

Имеет смысл постараться и подобрать функцию хеширования такой, чтобы в таблице не было коллизий. Поскольку заполнение таблицы происходит один раз, а доступ к ней производится многократно, эти затраты окупаются.

Таблица символов формируется динамически — в процессе работы 1-го прохода. Поиск же в этой таблице осуществляется как в 1-м проходе (перед занесением в таблицу нового имени, проверяется, нет ли его уже в таблице).

Построение этой таблицы как таблицы прямого доступа не очень целесообразно, так как неизбежно возникновение коллизий. Поэтому поиск в таблице символов может быть дихотомическим, но для этого таблица должна быть упорядочена.

Поскольку новые имена добавляются в таблицу «по одному», и после каждого добавления упорядоченность таблицы должна восстанавливаться, целесообразно применять алгоритму сортировки, чувствительные к исходной упорядоченности данных.

Эти же соображения относятся и к другим таблицам, формируемым Ассемблером в процессе работы. При больших размерах таблиц и размещении их на внешней памяти могут применяться и более сложные (но и более эффективные) методы их организации, например — B+-деревья.



Двухпроходный Ассемблер — второй проход

Обычно 2-й проход Ассемблера читает исходный модуль с самого начала и отчасти повторяет действия 1-го прохода (лексический разбор, распознавание команд и директив, подсчет адресов). Это, однако, скорее дань традиции — с тех времен, когда в вычислительных системах ощущалась нехватка (или даже полное отсутствие) внешней памяти. В те далекие времена колода перфокарт или рулон перфоленты, содержащие текст модуля, вставлялись в устройство ввода повторно. В системах с эволюционным развитием сохраняются перфокарты и перфоленты (виртуальные), так что схема работы Ассемблера — та же. В новых системах Ассемблер может создавать промежуточный файл — результат 1-го прохода, который является входом для 2-го прохода. Каждому оператору исходного модуля соответствует одна запись промежуточного файла, и формат этой записи приблизительно такой:

Признак команда/директива/комментарий	Адрес памяти	Номер команды/директивы в таблице команд/директив	Поле операндов	Текст исходного оператора

Текст исходного оператора нужен только для печати листинга, Ассемблер на 2-м проходе использует только первые 4 поля записи. Первое поле позволяет исключить строки, целиком состоящие из комментария. Второе поле позволяет избежать подсчета адресов, третье — поиска мнемоники в таблицах. Основная работа 2-го прохода состоит в разборе поля операндов и генерации объектного кода.

Некоторые общие соображения, касающиеся этой работы. Объектный код команды состоит из поля кода операции и одного или нескольких полей операндов. Код операции, как правило, имеет размер 1 байт, количество, формат и семантика полей операндом определяется для каждого типа команд данной аппаратной платформы. В общем случае операндом команды может быть:

- регистр;
- непосредственный операнд;

- адресное выражение.

Виды адресных выражений зависят от способов адресации вычислительной системы, некоторые (возможно, наиболее типовые) способы адресации:

- абсолютный адрес;
- [базовый регистр]+смещение (здесь и далее квадратные скобки означают «содержимое того, что взято в скобки»);
- [базовый регистр]+[индексный регистр]+смещение;
- имя+смещение;
- литерал.

Адресное выражение моет содержать арифметические операции, соображения, касающиеся арифметики в этом случае — те же, что и в адресной арифметике языка C.

Имена в адресных выражениях должны заменяться на значения. Замена абсолютных имен (определенных в директиве EQU) очень проста — значение имени из таблицы символов просто подставляется вместо имени. Перемещаемые имена (метки и имена переменных) превращаются Ассемблером в адресное выражение вида [базовый регистр]+смещение. В таблице символов значения этих имен определены как смещение соответствующих ячеек памяти относительно начала программы. При трансляции имен необходимо, чтобы:

- Ассемблер «знал», какой регистр он должен использовать в качестве базового;
- Ассемблер «знал», какое значение содержится в базовом регистре;
- в базовом регистре действительно содержалось это значение.

Первые два требования обеспечиваются директивами, третье — машинными командами. Ответственность за то, чтобы при обеспечении этих требований директивы согласовывались с командами, лежит на программисте. Эти требования по-разному реализуются в разных вычислительных системах. Приведем два примера.

В Intel Ассемблер использует в качестве базовых сегментные регистры (DS при трансляции имен переменных, CS при трансляции меток). Для простой программы, состоящей из одной секции,

Загрузчик перед выполнением заносит во все сегментные регистры сегментный адрес начала программы и Ассемблер считает все смещения относительно него.

Сложная программа может состоять из нескольких секций, и в сегментном регистре может содержаться адрес той или иной секции, причем содержимое сегментного регистра может меняться в ходе выполнения программы. Загрузка в сегментный регистр адреса секции выполняется машинными командами:

```
MOV AX,секция
MOV сегментный_регистр,AX
```

Для того, чтобы Ассемблер знал, что адрес секции находится в сегментном_регистре, применяется директива:

```
ASSUME сегментный_регистр:секция
```

Далее при трансляции имен Ассемблер превращает имена в адресные выражения вида

```
[сегментный_регистр]+смещение в секции
```

Отмена использования сегментного регистра задается директивой:

```
ASSUME сегментный_регистр:NOTHING
```

Обратим внимание на то, что при трансляции команды

```
MOV AX,секция
```

в поле операнда заносится относительный адрес секции, при выполнении же здесь должен быть ее абсолютный адрес. Поэтому поля операндов такого типа должны быть модифицированы Загрузчиком после размещения программы в оперативной памяти.

Более гибкая система базовой адресации применяется в S/360, S/370, S/390. В качестве базового может быть использован любой регистр общего назначения. Директива:

```
USING относительный_адрес,регистр
```

сообщает Ассемблеру, что он может использовать регистр в качестве базового, и в регистре содержится адрес — 1-й операнд. Чаще всего относительный_адрес кодируется как * (обозначение текущего значения счетчика адреса), это означает, что в регистре содержится адрес первой команды, следующей за директивой USING. Занесение адреса в базовый регистр выполняется машинной командой BALR. Обычный контекст определения базового регистра:

```
BALR регистр,0
USING *,регистр
```

С такими операндами команда BALR заносит в регистр адрес следующей за собой команды.

Зная смещение именованной ячейки относительно начала программы и смещение относительно начала же программы содержимого базового регистра, Ассемблер легко может вычислить смещение именованной ячейки относительно содержимого базового регистра.

В отличие от предыдущего примера, в этом случае не требуется модификации при загрузке, так как команда BALR заносит в регистр абсолютный адрес.

Директива

```
DROP регистр
```

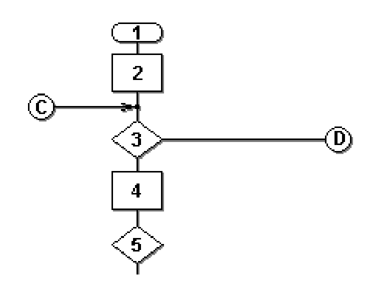
отменяет использование регистра в качестве базового.

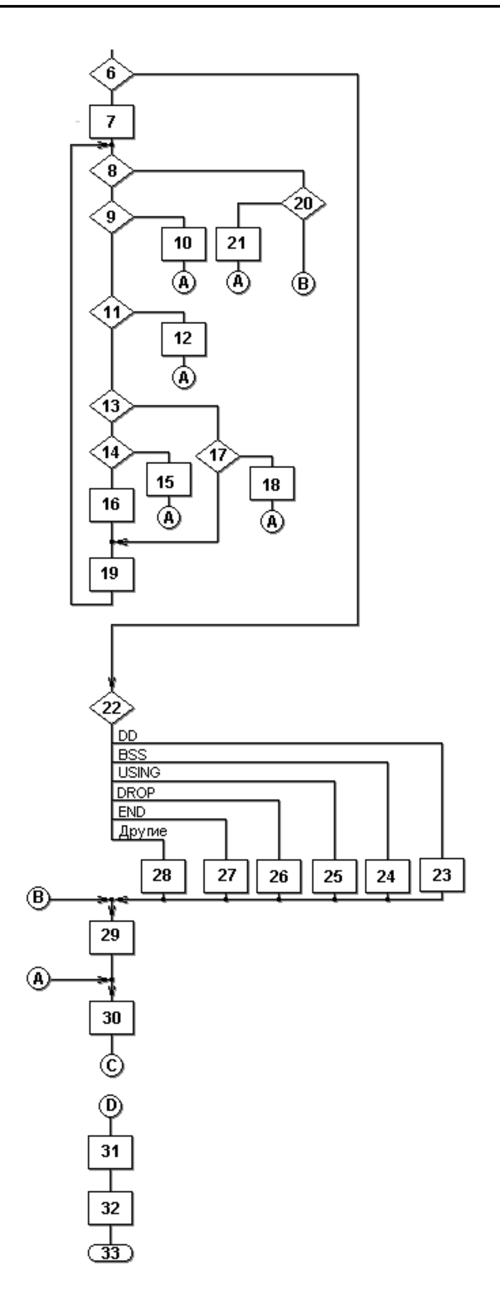
В качестве базовых могут быть назначены несколько регистров, Ассемблер сам выбирает, какой из них использовать в каждом случае.

Выше мы говорили, что Ассемблер «знает» базовый регистр и его содержимое. Это «знание» хранится в таблице базовых регистров. Обычно таблица содержит строки для всех регистров, которые могут быть базовыми и признак, используется ли регистр в таком качестве. Формат строки таблицы:

Регистр	Значение в регистре	Признак использования

Алгоритм выполнения 2-го прохода представлен на рисунке. Мы исходили из того, что 2-й проход использует промежуточный файл, сформированный 1-м проходом. Если же 2-й проход использует исходный модуль, то алгоритм должен быть расширен лексическим разбором оператора, распознаванием мнемоники и подсчетом адресов — так же, как и в 1-м проходе.





- **Блок1:** Начало 2-го прохода ассемблирования.
- **Блок2:** Начальные установки:
  - создание пустой таблицы базовых регистров;
  - открытие промежуточного файла исходного модуля;
  - установка в FASLE признака окончания
- **Блок3:** Признак окончания TRUE?
- **Блок4:** Считывание следующей записи промежуточного файла.
- **Блок5:** Если запись промежуточного файла описывает комментарий, переход на печать строки листинга.
- **Блок6:** Выясняется, содержит оператор команду или директиву
- **Блок7:** Если оператор содержит команду, формируется байт кода операции (код выбирается из таблицы команд) в объектном коде.
- **Блок8:** Выделение следующего элемента из списка операндов с удалением его из списка и с проверкой, не обнаружен ли при выделении конец списка операндов?
- **Блок9:** Если конец не обнаружен, обрабатывается выделенный операнд. Проверяется, не превысило ли число операндов требуемого для данного типа команды (выбирается из таблицы команд)
- **Блок10:** Если число операндов превышает требуемое — формирование сообщения об ошибке
- **Блок11:** Если число операндов правильное, распознается и проверяется тип операнда.
- **Блок12:** Если тип операнда не распознан или недопустим для данной команды — формирование сообщения об ошибке.
- **Блок13:** Есть ли в команде имя?
- **Блок14:** Если в команде есть имя, оно ищется в таблице символов.
- **Блок15:** Если имя в таблице символов не найдено — формирование сообщения об ошибке.
- **Блок16:** Если найдено имя в таблице символов, оно переводится в «база-смещение»
- **Блок17:** Если имени в команде нет, выполняется разбор и интерпретация операнда с проверкой правильности его кодирования.

- **Блок18:** Если обнаружены ошибки в кодировании операнда — формирование сообщения об ошибке.
- **Блок19:** Формируется код поля операнда и заносится в объектный код команды и обрабатывается следующий элемент списка операндов.
- **Блок20:** Если обнаружен конец списка операндов, проверяется, не меньше ли число операндов требуемого для данного типа команды. Если число операндов соответствует требуемого, управление переходит на вывод объектного кода.
- **Блок21:** Если число операндов меньше требуемого — формирование сообщения об ошибке
- **Блок22:** Если обрабатываемый оператор является директивой, алгоритм разветвляется, в зависимости от того, какая это директива. При обработке любой директивы производится разбор и анализ ее операндов и (не показано на схеме алгоритма) возможно формирование сообщения об ошибке.
- **Блок23:** Обработка директивы типа DD включает в себя:
  - выделение элементов списка операндов;
  - для каждого элемента — распознавание типа и значения константы;
  - генерация объектного кода константы;
  - обработка возможных коэффициентов повторения.
- **Блок24:** Обработка директивы типа BSS может вестись точно так же, как и DD за исключением того, что вместо кода константы генерируются некоторые «пустые» коды. Однако, эти коды не нужны в объектном модуле, они могут не генерироваться, в этом случае должны предприниматься некоторые действия, формирующие «разрыв» в объектных кодах.
- **Блок25:** Обработка директивы типа USING (ASSUME) включает в себя занесение в соответствующую строку таблицы базовых регистров значения операнда-адреса и установку для данного регистра признака использования.
- **Блок26:** Обработка директивы типа USING (ASSUME) включает в себя занесение в соответствующую строку таблицы базовых регистров значения операнда-адреса и установку для данного регистра признака использования.

- **Блок27:** Обработка директивы END устанавливает признак окончания в TRUE. При обработке этой директивы в объектный модуль также может заносится стартовый адрес программы — параметр директивы.
- **Блок28:** Обработка прочих директив ведется по своим алгоритмам.
- **Блок29:** После окончания обработки команды или директивы сформированный объектный код выводится в файл объектного модуля.
- **Блок30:** Печать строки листинга. На эту точку также управление передается при выявлении ошибок. При наличии ошибки сообщение об ошибке печатается после строки листинга. Управление затем передается на считывание следующей записи промежуточного файла.
- **Блок31:** После того, как установлен признак окончания работы формируются и выводятся в объектный модуль коды литерального пула, таблицы связываний и перемещений.
- **Блок32:** Закрываются файлы, освобождается выделенная память.
- **Блок33:** Работа Ассемблера завершается.

При рассмотрении алгоритма мы принимали во внимание только генерацию объектных кодов, соответствующих командам и константам. Мы не рассматривали то, какова общая структура объектного модуля.



Некоторые дополнительные директивы

OGR
Установка адреса

Операндом директивы является числовая константа или выражение, вычисляемое при ассемблировании. Как правило, Ассемблер считает, что первая ячейка обрабатываемой им программы располагается по адресу 0. Директива ORG устанавливает счетчик адресов программы в значение, определяемое операндом. Так, при создании COM-программ для MS DOS программа начинается с директивы ORG 100H. Этим оператором резервируется 256 байт в начале программы для размещения префикса программного сегмента.

В абсолютных программах директива применяется для размещения программы по абсолютным адресам памяти.

START/
SECT

Начало модуля или программной секции. Операндом директивы является имя секции. Этой директивой устанавливается в 0 счетчик адресов программы. Программа может состоять из нескольких программных секций, в каждой секции счет адресов ведется от 0. При обработке этой директивы на 1-м проходе Ассемблер создает таблицу программных секций.

На 1-м проходе Ассемблер составляет список секций, и только в конце 1-го прохода определяет их длины и относительные адреса в программе. На 2-м проходе Ассемблер использует таблицу секций при трансляции адресов.



Директивы связывания

ENT
Входная точка

Операндом этой директивы является список имен входных точек программного модуля — тех точек, на которые может передаваться управление извне модуля или тех данных, к которым могут быть обращения извне.

EXT
Внешняя точка

Операндом этой директивы является список имен, к которым есть обращение в модуле, но сами эти имена определены в других модулях.

Эти директивы обрабатываются на 2-м проходе, и на их основе строятся таблицы связываний и перемещений.



Одно- и многопроходный Ассемблер

Мы показали, что в двухпроходном Ассемблере на 1-м проходе осуществляется определение имен, а на втором — генерация кода.

Можно ли построить однопроходный Ассемблер? Трудность состоит в том, что в программе имя может появиться в поле операнда команды прежде, чем это имя появится в поле метки/имени, и Ассемблер не может преобразовать это имя в адресное выражение, та как еще на знает его значение. Как решить эту проблему?

Запретить ссылки вперед. Имя должно появляться в поле операнда только после того, как оно было определено как метка при команде или данных или через директиву EQU. В этом случае построить Ассемблер легко, но накладываются ограничения, стесняющие действия программиста.

Если объектный модуль сохраняется в объектной памяти, то Ассемблер может отложить формирование кода для операнда — неопределенного имени и вернуться к нему, когда имя будет определено. При появлении в поле операнда команды неопределенного имени поле операнда не формируется (заполняется нулями). Таблица символов расширяется полями: признаком определенного/неопределенного имени, и указателем на список адресов в объектном модуле, по которым требуется модификация поля операнда.

При появлении имени в поле операнда Ассемблер ищет имя в таблице символов. Если имя найдено и помечено как определенное, Ассемблер транслирует его в адресное выражение, как и при 2-проходном режиме. Если имя не найдено, Ассемблер заносит имя в таблицу символов, помечает его неопределенным и создает первый элемент связанного с именем списка, в который заносит адрес в объектном модуле операнда данной команды. Если имя найдено, но помечено как определенное, Ассемблер добавляет в список, связанный с данным именем, адрес в объектном модуле операнда данной команды.

При появлении имени в поле метки команды или директивы Ассемблер определяет значение имени и ищет его в таблице символов. Если имя не найдено, Ассемблер добавляет имя в таблицу и помечает его как определенное. Если имя найдено и помечено как определенное, Ассемблер выдает сообщение об ошибке (неуникальное имя). Если имя найдено, но помечено как определенное, Ассемблер обрабатывает связанный с данным именем список: для каждого элемента списка по хранящемуся в нем адресу в объектном модуле записывается значение имени. После обработки список уничтожается, значение имени сохраняется в таблице символов и имя помечается как определенное.

После окончания прохода Ассемблер проверяет таблицу символов: если в ней остались неопределенные имена, выдается сообщение об ошибке.

Если объектный модуль выводится в файл по мере его формирования, алгоритм работы Ассемблера очень похож на предыдущий случай. Ассемблер не может при определении имени исправить уже сформированные и выведенные в файл операнды, но вместо этого он формирует новую запись объектного модуля, исправляющую старую. При загрузке будет сначала загружена в память запись с пустым полем операнда, но затем на ее место — запись с правильным полем операнда.

Для чего может понадобиться многопроходный Ассемблер? Единственная необходимость во многих проходах может возникнуть, если в директиве EQU разрешены ссылки вперед. Мы упоминали выше, что имя в директиве EQU может определяться через другое имя. В одно- или двухпроходном Ассемблере это другое имя должно быть обязательно определено в программе выше. В многопроходном Ассемблере оно может быть определено и ниже. На первом проходе происходит определение имен и составление таблицы символов, но некоторые имена остаются неопределенными. На втором проходе определяются имена, не определившиеся во время предыдущего прохода. Второй проход повторяется до тех пор, пока не будут определены все имена (или не выяснится, что какие-то имена определить невозможно). На последнем проходе генерируется объектный код.

Лекция 11. Логика и организация программы



Команда JMP

Большинство программ содержат ряд циклов, в которых несколько команд повторяются до достижения определенного требования, и различные проверки, определяющие, какие из нескольких действий следует выполнять. Обычным требованием является проверка — должна ли программа завершить выполнение.

Эти требования включают передачу управления по адресу команды, которая не находится непосредственно за выполняемой в текущий момент командой. Передача управления может осуществляться вперед для выполнения новой группы команд или назад для повторения уже выполненных команд.

Некоторые команды могут передавать управление, изменяя нормальную последовательность шагов непосредственной модификацией значения смещения в командном указателе. Ниже приведены четыре способа передачи управления:

Безусловный переход
```
JMP
```
Цикл:
```
LOOP
```
Условный переход:
```
J nnn (больше,меньше,равно)
```
Вызов процедуры:
```
CALL
```


Заметим, что имеется три типа адресов: **SHORT**, **NEAR** и **FAR**. Адресация SHORT используется при циклах, условных переходах и некоторых безусловных переходах. Адресация NEAR и FAR используется для вызовов процедур (CALL) и безусловных переходов, которые не квалифицируются , как SHORT. Все три типа передачи управления воздействуют на содержимое регистра IP; тип FAR также изменяет регистр CS.

Одной из команд обычно используемых для передачи управления является команда JMP. Эта команда выполняет безусловный переход, то есть, обеспечивает передачу управления при любых обстоятельствах.



Команда LOOP

Команда JMP реализует бесконечный цикл. Но более вероятно подпрограмма должна выполнять определенное число циклов. Команда LOOP, которая служит для этой цели, использует начальное значение в регистре CX. В каждом цикле команда LOOP автоматически уменьшает содержимое регистра CX на 1. Пока значение в CX не равно нулю, управление передается по адресу, указанному в операнде, и если в CX будет 0, управление переходит на следующую после LOOP команду.

Аналогично команде JMP, операнд команды LOOP определяет расстояние от конца команды LOOP до адреса метки, которое прибавляется к содержимому командного указателя. Для команды LOOP это расстояние должно быть в пределах от -128 до +127 байт. В случае, если операнд превышает эти границы, то Ассемблер выдаст сообщение:

```
«Relative jump out of range» (превышены границы перехода)
```

Для проверки команды LOOP рекомендуется изменить соответствующим образом программу, выполнить ее ассемблирование, компоновку и преобразование в COM-файл. Для трассировки всех циклов используйте отладчик DEBUG. Когда в значение регистре CX уменьшится до нуля, содержимое регистров AX, BX и DX будет соответственно шест. 000B, 0042 и 0400. Для выхода из отладчика введите команду Q.

Дополнительно существует две разновидности команды LOOP — это LOOPE (или LOOPZ) и LOOPNE (или LOOPNZ). Обе команды также уменьшают значение регистра CX на 1. Команда LOOPE передает управление по адресу операнда, если регистр CX имеет ненулевое значение

и флаг нуля установлен (ZF=1). Команда LOOPNE передает управление по адресу операнда, если регистр CX имеет ненулевое значение и флаг нуля сброшен (ZF=0).



Флаговый регистр

Флаговый регистр содержит 16 бит флагов, которые управляются различными командами для индикации состояния операции. Во всех случаях флаги сохраняют свое значение до тех пор, пока другая команда не изменит его. Флаговый регистр содержит следующие девять используемых бит (звездочками отмечены неиспользуемые биты):

Номер бита:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Флаг:	*	*	*	*	O	D	I	T	S	Z	*	A	*	P	*	C

Рассмотрим эти флаги в последовательности справа налево.

CF (Carry Flag) — флаг переноса

Содержит значение «переносов» (0 или 1) из старшего разряда при арифметических операциях и некоторых операциях сдвига и циклического сдвига.

PF (Parity Flag) — флаг четности

Проверяет младшие восемь бит результатов операций над данными. Нечетное число бит приводит к установке этого флага в 0, а четное — в 1. Не следует путать флаг четности с битом контроля на четность.

AF (Auxiliary Carry Flag) — дополнительный флаг переноса

Устанавливается в 1, если арифметическая операция приводит к переносу четвертого справа бита (бит номер 3) в регистровой однобайтовой команде.Данный флаг имеет отношение к арифметическим операциям над символами кода ASCII и к десятичным упакованным полям.

ZF (Zero Flag) — флаг нуля

Устанавливается в качестве результата арифметических команд и команд сравнения. Как это ни странно, ненулевой результат приводит к установке нулевого значения этого флага, а нулевой — к установке единичного значения. Кажущееся несоответствие является, однако, логически правильным, так как 0 обозначает «нет» (то есть, результат не равен нулю), а единица обозначает «да» (то есть, результат равен нулю).

Команды условного перехода JE и JZ проверяют этот флаг.

SF (SIgn Flag) — знаковый флаг

Устанавливается в соответствии со знаком результата (старшего бита) после арифметических операций: положительный результат устанавливает 0, а отрицательный — 1. Команды условного перехода JG и JL проверяют этот флаг.

TF (Trap Flag) — флаг пошагового выполнения

Этот флаг вам уже приходилось устанавливать, когда использовалась команда T в отладчике DEBUG. В случае, если этот флаг установлен в единичное состояние, то процессор переходит в режим пошагового выполнения команд, то есть, в каждый момент выполняется одна команда под пользовательским управлением.

IF (Interrupt Flag) — флаг прерывания

При нулевом состоянии этого флага прерывания запрещены, при единичном — разрешены.

DF (DIrection Flag) — флаг направления

Используется в строковых операциях для определения направления передачи данных. При нулевом состоянии команда увеличивает содержимое регистров SI и DI, вызывая передачу данных слева направо, при нулевом — уменьшает содержимое этих регистров, вызывая передачу данных справа налево.

OF (Overflow Flag) — флаг переполнения

Фиксирует арифметическое переполнение, то есть, перенос вниз старшего (знакового) бита при знаковых арифметических операциях.

В качестве примера: команда CMP сравнивает два операнда и воздействует на флаги AF, CF, OF, PF, SF, ZF. Однако, нет необходимости проверять все эти флаги по отдельности. В следующем примере проверяется содержит ли регистр BX нулевое значение:

```
CMP BX,00 ;Сравнение BX с нулем
JZ B50 ;Переход на B50 если нуль
...
действия не при нуле
...
B50: ... ;Точка перехода при BX=0
```

В случае, если BX содержит нулевое значение, команда CMP устанавливает флаг нуля ZF в единичное состояние, и возможно изменяет (или нет) другие флаги.

Команда JZ (переход, если нуль) проверяет только флаг ZF. При единичном значении ZF, обозначающее нулевой признак, команда передает управление на адрес, указанный в ее операнде, то есть, на метку B50.



Команды условного перехода

Команда LOOP уменьшает на единицу содержимое регистра CX и проверяет его: если не ноль, то управление передается по адресу, указанному в операнде. Таким образом, передача управления зависит от конкретного состояния. Ассемблер поддерживает большое количество команд условного перехода, которые осуществляют передачу управления в зависимости от состояний флагового регистра. Например, при сравнении содержимого двух полей последующий переход зависит от значения флага.

Команду LOOP в программе можно заменить на две команды: одна уменьшает содержимое регистра CX, а другая выполняет условный переход:

```
LOOP A20
DEC CX
JNZ A20
```

Команды DEC и JNZ действуют аналогично команде LOOP: уменьшают содержимое регистра CX на 1 и выполняет переход на метку A20, если в CX не ноль. Команда DEC кроме того устанавливает флаг нуля во флаговом регистре в состояние 0 или 1. Команда JNZ затем проверяет эту установку. В рассмотренном примере команда LOOP хотя и имеет ограниченное использование, но более эффективна, чем две команды: DEC и JNZ.

Аналогично командам JMP и LOOP операнд в команде JNZ содержит значение расстояния между концом команды JNZ и адресом A20, которое прибавляется к командному указателю. Это расстояние должно быть в пределах от -128 до +127 байт.

В случае перехода за эти границы Ассемблер выдаст сообщение:

```
«Relative jump out of range» (превышены относительные границы перехода)
```

Знаковые и беззнаковые данные

Рассматривая назначение команд условного перехода следует пояснить характер их использования. Типы данных, над которыми выполняются арифметические операции и операции сравнения определяют какими командами пользоваться: беззнаковыми или знаковыми. Беззнаковые данные используют все биты как биты данных; характерным примером являются символьные строки: имена, адреса и натуральные числа. В знаковых данных самый левый бит представляет собой знак, причем если его значение равно нулю, то число положительное, и если единице, то отрицательное. Многие числовые значения могут быть как положительными так и отрицательными.

В качестве примера предположим, что регистр AX содержит 11000110, а BX — 00010110. Команда:

```
CMP AX,BX
```

сравнивает содержимое регистров AX и BX. В случае, если данные беззнаковые, то значение в AX больше, а если знаковые — то меньше.

Запомните, что для беззнаковых данных есть переходы по состояниям равно, выше или ниже, а для беззнаковых — равно, больше или меньше. Переходы по проверкам флагов переноса, переполнения и паритета имеют особое назначение. Ассемблер транслирует мнемонические коды в объектный код независимо от того, какую из двух команд вы применили.



Процедуры и оператор CALL

Ранее примеры содержали в кодовом сегменте только одну процедуру, оформленную следующим образом:

```
BEGIN PROC FAR
.
.
BEGIN ENDP
```

Операнд FAR информирует систему о том, что данный адрес является точкой входа для выполнения, а директива ENDP определяет конец процедуры.

Кодовый сегмент, однако, может содержать любое количество процедур, которые разделяются директивами PROC и ENDP.

Обратите внимание на следующие особенности:

- Директивы PROC по меткам имеют операнд NEAR для указания того, что эти процедуры находятся в текущем кодовом сегменте.
- Каждая процедура имеет уникальное имя и содержит собственную директиву ENDP для указания конца процедуры.
- Для передачи управления в процедуре BEGIN имеются две команды: CALL. В результате первой команды CALL управление передается указанной процедуре и начинается ее выполнение. Достигнув команды RET, управление возвращается на команду непосредственно следующую за первой командой CALL. Вторая команда CALL действует аналогично — передает управление в указанную процедуру, выполняет ее команды и возвращает управление по команде RET.
- Команда RET всегда выполняет возврат в вызывающую программу.

Программа BEGIN вызывает процедуры, которые возвращают управление обратно в BEGIN. Для выполнения самой программы BEGIN операционная система DOS вызывает ее и в конце выполнения команда RET возвращает управление в DOS. В случае, если процедура не содержит завершающей команды RET, то выполнение команд продолжится непосредственно в этой процедуре. В случае, если процедура не содержит команды RET, то будут выполняться команды, оказавшиеся за процедурой с непредсказуемым результатом.

Использование процедур дает хорошую возможность организовать логическую структуру программы. Кроме того, операнды для команды CALL могут иметь значения, выходящие за границу от -128 до +127 байт.

Технически управление в процедуру типа NEAR может быть передано с помощью команд перехода или даже обычным построчным кодированием. Но в большинстве случаев рекомендуется использовать команду CALL для передачи управления в процедуру и команду RET для возврата.



Сегмент стека

Ниже буду рассмотрены только две команды, использующие стек, — это команды PUSH в начале сегмента кодов, которые обеспечивают возврат в DOS, когда EXE-программа завершается.

Естественно для этих программ требуется стек очень малого размера. Однако, команда CALL автоматически записывает в стек относительный адрес команды, следующей непосредственно за командой CALL, и увеличивает после этого указатель вершины стека. В вызываемой процедуре команда RET использует этот адрес для возврата в вызывающую процедуру и при этом автоматически уменьшается указатель вершины стека.

Таким образом, команды PUSH записывают в стек двухбайтовые адреса или другие значения. Команды POP обычно выбирают из стека записанные в него слова. Эти операции изменяют относительный адрес в регистре SP (то есть, в указатели стека) для доступа к следующему слову. Данное свойство стека требует чтобы команды RET и CALL соответствовали друг другу. Кроме того, вызванная процедура может вызвать с помощью команды CALL другую процедуру, а та в свою очередь — следующую. Стек должен иметь достаточные размеры для того, чтобы хранить все записываемые в него адреса. Как правило, стек объемом в 32 слова является достаточным.

Команды PUSH, PUSHF, CALL, INT, и INTO заносят в стек адрес возврата или содержимое флагового регистра. Команды POP, POPF, RET и IRET извлекают эти адреса или флаги из стека.

При передаче управления в EXE-программу система устанавливает в регистрах следующие значения:

DS и ES

Адрес префикса программного сегмента — область в 256 (шест. 100) байт, которая предшествует выполняемому программному модулю в памяти.

CS

Адрес точки входа в программу (адрес первой выполняемой команды).

IP

Нуль.

SS

Адрес сегмента стека.



Команды логических операций: AND, OR, XOR, TEST, NOT

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования. Команды AND, OR, XOR и TEST — являются командами логических операций. Эти команды используются для сброса и установки бит и для арифметических операций в коде ASCII. Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги CF, OF, PF, SF, ZF.

AND

В случае, если оба из сравниваемых битов равны 1, то результат равен 1; во всех остальных случаях результат — 0.

OR

В случае, если хотя бы один из сравниваемых битов равен 1, то результат равен 1; если сравниваемые биты равны 0, то результат — 0.

XOR

В случае, если один из сравниваемых битов равен 0, а другой равен 1, то результат равен 1; если сравниваемые биты одинаковы (оба — 0 или оба — 1) то результат — 0.

TEST

Действует как AND — устанавливает флаги, но не изменяет биты.

Первый операнд в логических командах указывает на один байт или слово в регистре или в памяти и является единственным значением, которое может изменятся после выполнения команд. В следующих командах AND, OR и XOR используются одинаковые битовые значения:

```
AND OR XOR 0101 0101 0101 0011 0011 0011
```

Результат:

```
0001 0111 0110
```

Для следующих несвязанных примеров, предположим, что AL содержит 1100 0101, а BH содержит 0101 1100:

```
1. AND AL,BH ;Устанавливает в AL 0100 0100
2. OR BH,AL ;Устанавливает в BH 1101 1101
3. XOR AL,AL ;Устанавливает в AL 0000 0000
4. AND AL,00 ;Устанавливает в AL 0000 0000
5. AND AL,0FH ;Устанавливает в AL 0000 0101
6. OR CL,CL ;Устанавливает флаги SF и ZF
```

Примеры 3 и 4 демонстрируют способ очистки регистра. В примере 5 обнуляются левые четыре бита регистра AL. Хотя команды сравнения CMP могут быть понятнее, можно применить команду OR для следующих целей:

```
1. OR CX,CX ;Проверка CX на нуль JZ ... ;Переход, если нуль
2. OR CX,CX ;Проверка знака в CX JS ... ;Переход, если
отрицательно
```

Команда TEST действует аналогично команде AND, но устанавливает только флаги, а операнд не изменяется. Ниже приведено несколько примеров:

```
1. TEST BL,11110000B ;Любой из левых бит в BL JNZ ... ; равен
единице?
2. TEST AL,00000001B ;Регистр AL содержит JNZ ... ; нечетное
значение?
3. TEST DX,0FFH ;Регистр DX содержит JZ ... ; нулевое значение?
```

Еще одна логическая команда NOT устанавливает обратное значение бит в байте или в слове, в регистре или в памяти: нули становятся единицами, а единицы — нулями. В случае, если, например, регистр AL содержит 1100 0101, то команда NOT AL изменяет это значение на 0011 1010. Флаги не меняются.

Команда NOT не эквивалентна команде NEG, которая меняет значение с положительного на отрицательное и наоборот, посредством замены бит на противоположное значение и прибавления единицы.



Изменение строчных букв на заглавные

Существуют различные причины для преобразований между строчными и заглавными буквами. Например, вы могли получить файл данных, созданный на компьютере, который работает только с заглавными буквами. Или некая программа должна позволить пользователям вводить команды как заглавными, так и строчными буквами (например, YES или yes) и преобразовать их в заглавные для проверки. Заглавные буквы от A до Z имеют шест. коды от 41 до 5A, а строчные буквы от a до z имеют шест. коды от 61 до 7A.

Единственная разница в том, что пятый бит равен 0 для заглавных букв и 1 для строчных:

```
Буква A: 01000001
Буква a: 01100001
Буква Z: 01011010
Буква z: 01111010
```



Команды сдвига и циклического сдвига

Команды сдвига и циклического сдвига, которые представляют собой часть логических возможностей компьютера, имеют следующие свойства:

- обрабатывают байт или слово;
- имеют доступ к регистру или к памяти;
- сдвигают влево или вправо;
- сдвигают на величину до 8 бит (для байта) и 16 бит (для слова)
- сдвигают логически (без знака) или арифметически (со знаком).

Значение сдвига на 1 может быть закодировано как непосредственный операнд, значение больше 1 должно находиться в регистре CL.

Команды сдвига

При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита.

Существуют следующие команды сдвига:

```
SHR ;Логический (беззнаковый) сдвиг вправо
SHL ;Логический (беззнаковый) сдвиг влево
SAR ;Арифметический сдвиг вправо
SAL ;Арифметический сдвиг влево
```

Следующий фрагмент иллюстрирует выполнение команды SHR:

```
MOV CL,03 ; AX:
MOV AX,10110111B ; 10110111
SHR AX,1 ; 01011011 ;Сдвиг вправо на 1
SHR AX,CL ; 00001011 ;Сдвиг вправо на 3
```

Первая команда SHR сдвигает содержимое регистра AX вправо на 1 бит.

Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра AX заполняется нулем. Вторая команда сдвигает содержимое регистра AX еще на три бита. При этом флаг CF последовательно принимает значения 1, 1, 0, а в три левых бита в регистре AX заносятся нули.

Рассмотрим действие команд арифметического вправо SAR:

```
MOV CL,03 ; AX:
MOV AX,10110111B ; 10110111
SAR AX,1 ; 11011011 ;Сдвиг вправо на 1
SAR AX,CL ; 11111011 ;Сдвиг вправо на 3
```

Команда SAR имеет важное отличие от команды SHR: для заполнения левого бита используется знаковый бит. Таким образом, положительные и отрицательные величины сохраняют свой знак. В приведенном примере знаковый бит содержит единицу.

При сдвигах влево правые биты заполняются нулями. Таким образом, результат команд сдвига SHL и SAL индентичен.

Сдвиг влево часто используется для удваивания чисел, а сдвиг вправо — для деления на 2. Эти операции осуществляются значительно быстрее, чем команды умножения или деления. Деление пополам нечетных чисел (например, 5 или 7) образует меньшие значения (2 или 3, соответственно) и устанавливает флаг CF в 1. Кроме того, если необходимо

выполнить сдвиг на 2 бита, то использование двух команд сдвига более эффективно, чем использование одной команды с загрузкой регистра CL значением 2.

Для проверки бита, занесенного в флаг CF используется команда JC (переход, если есть перенос).

Команды циклического сдвига

Циклический сдвиг представляет собой операцию сдвига, при которой выдвинутый бит занимает освободившийся разряд. Существуют следующие команды циклического сдвига:

```
ROR ;Циклический сдвиг вправо
ROL ;Циклический сдвиг влево
RCR ;Циклический сдвиг вправо с переносом
RCL ;Циклический сдвиг влево с переносом
```

Следующая последовательность команд иллюстрирует операцию циклического сдвига ROR:

```
MOV CL,03 ; BX:
MOV BX,10110111B ; 10110111
ROR BX,1 ; 11011011 ;Сдвиг вправо на 1
ROR BX,CL ; 01111011 ;Сдвиг вправо на 3
```

Первая команда ROR при выполнении циклического сдвига переносит правый единичный бит регистра BX в освободившуюся левую позицию. Вторая команда ROR переносит таким образом три правых бита.

В командах RCR и RCL в сдвиге участвует флаг CF. Выдвигаемый из регистра бит заносится в флаг CF, а значение CF при этом поступает в освободившуюся позицию.

Рассмотрим пример, в котором используются команды циклического и простого сдвига. Предположим, что 32-битовое значение находится в регистрах DX:AX так, что левые 16 бит лежат в регистре DX, а правые — в AX. Для умножения на 2 этого значения возможны следующие две команды:

```
SHL AX,1 ;Умножение пары регистров
RCL DX,1 ; DX:AX на 2
```

Здесь команда SHL сдвигает все биты регистра AX влево, причем самый левый бит попадает в флаг CF. Затем команда RCL сдвигает все биты регистра DX влево и в освободившийся правый бит заносит значение из флага CF.



Организация программ

Ниже даны основные рекомендации для написания ассемблерных программ:

1. Четко представляйте себе задачу, которую должна решить программа.

2. Сделайте эскиз задачи в общих чертах и спланируйте общую логику программы.

Например, если необходимо проверить операции пересылки нескольких байт, начните с определения полей с пересылаемыми данными.

Затем спланируйте общую стратегию для инициализации, условного перехода и команды LOOP.

Приведем основную логику, которую используют многие программисты в таком случае:

```
инициализация стека и сегментных регистров
вызов подпрограммы цикла
возврат
```

Подпрограмма цикла может быть спланирована следующим образом:

```
инициализация регистров значениями адресов и числа циклов
Метка: пересылка одного байта увеличение адресов на 1 уменьшение
счетчика на 1: если счетчик не ноль, то идти на метку если ноль,
возврат
```

3. Представьте программу в виде логических блоков, следующих друг за другом. Процедуры не превышающие 25 строк (размер экрана) удобнее для отладки.

4. Пользуйтесь тестовыми примерами программ. Попытки запомнить все технические детали и программирование сложных программ «из головы» часто приводят к многочисленным ошибкам.

5. Используйте комментарии для описания того, что должна делать процедура, какие арифметические действия или операции сравнения будут выполняться и что делают редко используемые команды. (Например, команда XLAT, не имеющая операндов).

6. Для кодирования программы используйте заготовку программы, скопированной в файл с новым именем.

Важно:

- Метки процедур должны завершаться двоеточием для указания типа NEAR. Отсутствие двоеточия приводит к ассемблерной ошибке.
- Метки для команд условного перехода и LOOP должны лежать в границах -128 до +127 байт. Операнд таких команд генерирует один байт объектного кода. Шест. от 01 до 7F соответствует десятичным значениям от +1 до +127, а шест. от FF до 80 покрывает значения от -1 до +128. Так как длина машинной команды может быть от 1 до 4 байт, то соблюдать границы не просто. Практически можно ориентироваться на размер в два экрана исходного текста (примерно 50 строк).
- При использовании команды LOOP, инициализируйте регистр CX положительным числом. Команда LOOP контролирует только нулевое значение, при отрицательном программа будет продолжать циклиться.
- В случае, если некоторая команда устанавливает флаг, то данный флаг сохраняет это значение, пока другая команда его не изменит. Например, если за арифметической командой, которая устанавливает флаги, следуют команды MOV, то они не изменят флаги. Однако, для минимизации числа возможных ошибок, следует кодировать команды условного перехода непосредственно после команд, устанавливающих проверяемые флаги.
- Выбирайте команды условного перехода соответственно операциям над знаковыми или беззнаковыми данными.
- Для вызова процедуры используйте команду CALL, а для возврата из процедуры — команду RET. Вызываемая процедура может, в свою очередь, вызвать другую процедуру, и если следовать существующим соглашениям, то команда RET всегда будет выбирать из стека правильный адрес возврата.

- Будьте внимательны при использовании индексных операндов. Сравните:
  ```
  MOV AX,SI
  MOV AX,[SI]
  ```
  Первая команда MOV пересылает в регистр AX содержимое регистра SI. Вторая команда MOV для доступа к пересылаемому слову в памяти использует относительный адрес в регистре SI.
- Используйте команды сдвига для удваивания значений и для деления пополам, но при этом внимательно выбирайте соответствующие команды для знаковых и беззнаковых данных.

Лекция 12. Компоновка программ



Межсегментные вызовы

Примеры программ, рассматриваемых ранее, состояли из одного шага ассемблирования. Возможно, однако, выполнение программного модуля, состоящего из нескольких ассемблированных программ. В этом случае программу можно рассматривать, как состоящую из основной программы и одной или более подпрограмм. Причины такой организации программ состоят в следующем:

- бывает необходимо скомпоновать программы, написанные на разных языках, например, для объединения мощности языка высокого уровня и эффективности Ассемблера;
- программа, написанная в виде одного модуля, может оказаться слишком большой для ассемблирования;
- отдельные части программы могут быть написаны разными группами программистов, ассемблирующих свои модули раздельно;
- ввиду возможно большого размера выполняемого модуля, может появиться необходимость перекрытия частей программы в процессе выполнения.

Каждая программа ассемблируется отдельно и генерирует собственный уникальный объектный (OBJ) модуль. Программа компоновщик (LINK) затем компонует объектные модули в один объединенный выполняемый (EXE) модуль.

Обычно выполнение начинается с основной программы, которая вызывает одну или более подпрограмм. Подпрограммы, в свою очередь, могут вызывать другие подпрограммы.

Существует много разновидностей организации подпрограмм, но любая организация должна быть «понятна» и Ассемблеру, и компоновщику, и этапу выполнения. Следует быть внимательным к ситуациям, когда, например, подпрограмма 1 вызывает подпрограмму 2, которая вызывает подпрограмму 3 и, которая в свою очередь вызывает подпрограмму 1. Такой процесс, известный как рекурсия, может использоваться на практике, но при неаккуратном обращении может вызвать любопытные ошибки при выполнении.

Команды CALL используются для внутрисегментных вызовов, то есть, для вызовов внутри одного сегмента. Внутрисегментный CALL может быть короткий (в пределах от +127 до -128 байт) или длинный (превышающий указанные границы). В результате такой операции «старое» значение в регистре IP запоминается в стеке, а «новый» адрес перехода загружается в этот регистр.

Например, внутрисегментный CALL может иметь следующий объектный код: E82000. Шест.E8 представляет собой код операции, которая заносит 2000 в виде относительного адреса 0020 в регистр IP. Затем процессор объединяет текущий адрес в регистре CS и относительный адрес в регистре IP для получения адреса следующей выполняемой команды. При возврате из процедуры команда RET восстанавливает из стека старое значение в регистре IP и передает управление таким образом на следующую после CALL команду.

Вызов в другой кодовый сегмент представляет собой межсегментный (длинный) вызов. Данная операция сначала записывает в стек содержимое регистра CS и заносит в этот регистр адрес другого сегмента, затем записывает в стек значение регистра IP и заносит новый относительный адрес в этот регистр.

Таким образом в стеке запоминаются и адрес кодового сегмента и смещение для последующего возврата из подпрограммы.

Например, межсегментный CALL может состоять из следующего объектного кода:

```
9A 0002 AF04
```

Шест.9A представляет собой код команды межсегментного вызова которая записывает значение 0002 в виде 0200 в регистр IP, а значение AF04 в виде 04AF в регистр CS. Комбинация этих адресов указывает на первую выполняемую команду в вызываемой подпрограмме:

```
Кодовый сегмент 04AF0
Смещение в IP 0200
Действительный адрес 04CF0
```

При выходе из вызванной процедуры межсегментная команда возврата REP восстанавливает оба адреса в регистрах CS и IP и таким образом передает управление на следующую после CALL команду.



Атрибуты EXTRN и PUBLIC

Директива EXTRN

Директива EXTRN имеет следующий формат:

```
EXTRN имя:тип [, ... ]
```

Можно определить более одного имени (до конца строки) или закодировать дополнительные директивы EXTRN.

В другом ассемблерном модуле соответствующее имя должно быть определено и идентифицировано как PUBLIC.

Тип элемента может быть ABS, BYTE, DWORD, FAR, NEAR, WORD. Имя может быть определено через EQU и должно удовлетворять реальному определению имени.

Директива PUBLIC

Директива PUBLIC указывает Ассемблеру и компоновщику, что адрес указанного идентификатора доступен из других программ. Директива имеет следующий формат:

```
PUBLIC идентификатор [, ... ]
```

Можно определить более одного идентификатора (до конца строки) или закодировать дополнительные директивы PUBLIC. Идентификаторы могут быть метками (включая PROC-метки), переменными или числами.

Неправильными идентификаторами являются имена регистров и EQU-идентификаторы, определяющие значения более двух байт.



Компоновка программ на языке С и Ассемблере

Трудность описания связи программ на языке С и ассемблерных программ состоит в том, что различные версии языка С имеют разные соглашения о связях и для более точной информации следует пользоваться руководством по имеющейся версии языка С. Здесь приведем лишь некоторые соображения, представляющие интерес:

- Большинство версий языка С обеспечивают передачу параметров через стек в обратной (по сравнению с другими языками) последовательности. Обычно доступ, например, к двум параметрам, передаваемым через стек, осуществляется следующим образом:
  ```
  MOV ES,BP MOV BP,SP
  MOV DH,[BP+4]
  MOV DL,[BP+6] ...
  POP BP RET
  ```
- Некоторые версии языка С различают прописные и строчные буквы, поэтому имя ассемблерного модуля должно быть представлено в том же символьном регистре, какой используют для ссылки С-программы.
- В некоторых версиях языка С требуется, чтобы ассемблерные программы, изменяющие регистры DI и SI, записывали их содержимое в стек при входе и восстанавливали эти значения из стека при выходе.
- Ассемблерные программы должны возвращать значения, если это необходимо, в регистре AX (одно слово) или в регистровой паре DX:AX (два слова).
- Для некоторых версий языка С, если ассемблерная программа устанавливает флаг DF, то она должна сбросить его командой CLD перед возвратом.



Выполнение COM-программы

В отличие от EXE-файла, COM-файл не содержит заголовок на диске. Так как организация COM-файла намного проще, то для DOS необходимо «знать» только то, что тип файла — COM.

Загруженным в память COM- и EXE-файлам предшествует префикс программного сегмента. Первые два байта этого префикса содержат команду INT 20H (возврат в DOS). При загрузке COM-программы DOS устанавливает в четырех сегментных регистрах адрес первого байта PSP.

Затем устанавливается указатель стека на конец 64 Кбайтового сегмента (шест.FFFE) или на конец памяти, если сегмент не достаточно большой. В вершину стека заносится нулевое слово. В командный указатель помещается шест.100 (размер PSP). После этого управление передается по адресу регистровой пары CS:IP, то есть, на адрес непосредственно после PSP. Этот адрес является началом выполняемой COM-программы и должен содержать выполнимую команду.

При выходе из программы команда RET заносит в регистр IP нулевое слово, которое было записано в вершину стека при инициализации. В этом случае в регистровой паре CS:IP получается адрес первого байта PSP, где находится команда INT 20H. При выполнении этой команды управление передается в резидентную часть COMMAND.COM. (В случае, если программа завершается по команде INT 20H вместо RET, то управление непосредственно передается в COMMAND.COM).



Выполнение EXE-программы

EXE-модуль, созданный компоновщиком, состоит из следующих двух частей: 1) заголовок — запись, содержащая информацию по управлению и настройке программы и 2) собственно загрузочный модуль.

В заголовке находится информация о размере выполняемого модуля, области загрузки в памяти, адресе стека и относительных смещениях, которые должны заполнить машинные адреса в соответствии с относительными шест.позициями:

00 Шест.4D5A

Компоновщик устанавливает этот код для идентификации правильного EXE-файла.

02

Число байтов в последнем блоке EXE-файла.

04

Число 512 байтовых блоков EXE-файла, включая заголовок.

06

Число настраиваемых элементов.

08

Число 16-байтовых блоков (параграфов) в заголовке, (необходимо для локализации начала выполняемого модуля, следующего после заголовка).

0A

Минимальное число параграфов, которые должны находится после загруженной программы.

0C

Переключатель загрузки в младшие или старшие адреса. При компоновке программист должен решить, должна ли его программа загружаться для выполнения в младшие адреса памяти или в старшие Обычным является загрузка в младшие адреса. Значение шест.0000 указывает на загрузку в старшие адреса, а шест.FFFF — в младшие. Иные значения определяют максимальное число параграфов, которые должны находиться после загруженной программы.

0E

Относительный адрес сегмента стека в выполняемом модуле.

10

Адрес,который загрузчик должен поместить в регистр SP перед передачей управления в выполнимый модуль.

12

Контрольная сумма — сумма всех слов в файле (без учета переполнений) используется для проверки потери данных.

14

Относительный адрес, который загрузчик должен поместить в регистр IP до передачи управления в выполняемый модуль.

16

Относительный адрес кодового сегмента в выполняемом модуле. Этот адрес загрузчик заносит в регистр CS.

18

Смещение первого настраиваемого элемента в файле.

1A

Номер оверлейного фрагмента: нуль обозначает, что заголовок относится к резидентной части EXE-файла.

1C

Таблица настройки, содержащая переменное число настраиваемых элементов, соответствующее значению по смещению 06.

Заголовок имеет минимальный размер 512 байтов и может быть больше, если программа содержит большое число настраиваемых элементов. Позиция 06 в заголовке указывает число элементов в выполняемом модуле, нуждающихся в настройке. Каждый элемент настройки в таблице, начинающейся в позиции 1C заголовка, состоит из двухбайтовых величин смещений и двухбайтовых сегментных значений.

Система строит префикс программного сегмента следом за резидентной частью COMMAND.COM, которая выполняет операцию загрузки. Затем COMMAND.COM выполняет следующие действия:

- Считывает форматированную часть заголовка в память.
- Вычисляет размер выполнимого модуля (общий размер файла в позиции 04 минус размер заголовка в позиции 08) и загружает модуль в память с начала сегмента.
- Считывает элементы таблицы настройки в рабочую область и прибавляет значения каждого элемента таблицы к началу сегмента (позиция OE).
- Устанавливает в регистрах SS и SP значения из заголовка и прибавляет адрес начала сегмента.

- Устанавливает в регистрах DS и ES сегментный адрес префикса программного сегмента.
- Устанавливает в регистре CS адрес PSP и прибавляет величину смещения в заголовке (позиция 16) к регистру CS. В случае, если сегмент кода непосредственно следует за PSP, то смещение в заголовке равно 256 (шест.100). Регистровая пара CS:IP содержит стартовый адрес в кодовом сегменте, то есть, начальный адрес программы.

После инициализации регистры CS и SS содержат правильные адреса, а регистр DS (и ES) должны быть установлены в программе для их собственных сегментов данных:

```
1. PUSH DS ;Занести адрес PSP в стек
2. SUB AX,AX ;Занести нулевое значение в стек
3. PUSH AX ; для обеспечения выхода из программы
4. MOV AX,datasegname ;Установка в регистре DX
5. MOV DS,AX ; адреса сегмента данных
```

При завершении программы команда RET заносит в регистр IP нулевое значение, которое было помещено в стек в начале выполнения программы. В регистровой паре CS:IP в этом случае получается адрес, который является адресом первого байта PSP, где расположена команда INT 20H. Когда эта команда будет выполнена, управление перейдет в DOS.



Функции загрузки и выполнения программы

Рассмотрим теперь, как можно загрузить и выполнить программу из другой программы. Функция шест.4B дает возможность одной программе загрузить другую программу в память и при необходимости выполнить.

Для этой функции необходимо загрузить адрес ASCIIZ-строки в регистр DX, а адрес блока параметров в регистр BX (в действительности в регистровую пару ES:BX). В регистре AL устанавливается номер функции 0 или 3:

AL=0. Загрузка и выполнение

Данная операция устанавливает префикс программного сегмента для новой программы, а также адрес подпрограммы реакции на Cntrl/Break и адрес передачи управления на следующую команду после завершения новой программы. Так как все регистры, включая SP, изменяют свои значения, то данная операция не для новичков.

Блок параметров, адресуемый по ES:BX, имеет следующий формат:

0

Двухбайтовый сегментный адрес строки параметров для передачи.

2

Четырехбайтовый указатель на командную строку в PSP+80H.

6

Четырехбайтовый указатель на блок FCB в PSP+5CH.

10

Четырехбайтовый указатель на блок FCB в PSP+6CH.

AL=3. Оверлейная загрузка

Данная операция загружает программу или блок кодов, но не создает PSP и не начинает выполнение.

Таким образом можно создавать оверлейные программы. Блок параметров адресуется по регистровой паре ES:BX и имеет следующий формат:

0

Двухбайтовый адрес сегмента для загрузки файла.

2

Двухбайтовый фактор настройки загрузочного модуля.

Возможные коды ошибок, возвращаемые в регистре AX: 01, 02, 05, 08, 10 и 11.

Важно:

- В основной программе, вызывающей подпрограмму, необходимо определять точку входа как EXTRN, а в подпрограмме — как PUBLIC.

- Будьте внимательны при использовании рекурсий, когда подпрограмма 1 вызывает подпрограмму 2, которая в свою очередь вызывает подпрограмму 1.
- В случае, если кодовые сегменты необходимо скомпоновать в один сегмент, то необходимо определить их с одинаковыми именами, одинаковыми классами и атрибутом PUBLIC.
- Для простоты программирования начинайте выполнение с основной программы.
- Определение общих данных в основной программе обычно проще (но не обязательно). Основная программа определяет общие данные как PUBLIC, а подпрограмма (или подпрограммы) — как EXTRN.

Лекция 13. Выполнение программ



Начинаем работать

Когда рабочая часть DOS будет загружена в память, на экране появится запрос для ввода даты и времени, а затем буква текущего дисковода, обычно A для дискеты и C для винчестера (твердого диска). Изменить текущий дисковод можно, нажав соответствующую букву, двоеточие и клавишу **Enter**.

Это обычная процедура загрузки, которую следует использовать всякий раз.

В этом первом упражнении для просмотра содержимого ячеек памяти используется программа DOS DEBUG. Для запуска этой программы введите DEBUG и нажмите **Enter**, в результате программа DEBUG должна загрузится с диска в память. После окончания загрузки на экране появится приглашение в виде дефиса, что свидетельствует о готовности программы DEBUG для приема команд.

Размер памяти

Сначала проверим размер доступной для работы памяти. В зависимости от модели компьютера это значение связано с установкой внутренних переключателей и может быть меньше, чем реально существует. Данное значение находится в ячейках памяти шест.413 и 414 и его можно просмотреть из DEBUG по адресу, состоящему из двух частей: 400 — это адрес сегмента, который записывается как 40 (последний нуль подразумевается) и 13 — это смещение от начала сегмента. Таким образом, можно ввести следующий запрос:

```
D 40:13 (и нажать Enter).
```


Первые два байта, появившиеся в результате на экране, содержат размер памяти в килобайтах и в шестнадцатеричном представлении, причем байты располагаются в обратной последовательности.

Серийный номер

Серийный номер компьютера «зашит» в ROM по адресу шест. FE000. Для того, чтобы увидеть его, следует ввести:

```
D FE00:0 (и нажать Enter)
```

В результате на экране появится семизначный номер компьютера и дата копирайт.

Дата ROM BIOS

Дата ROM BIOS в формате **mm/dd/yy** находится по шест. адресу FFFF5. Введите

```
D FFFF:05 (и нажмите Enter)
```

Знание этой информации (даты) иногда бывает полезным для определения модели и возраста компьютера. Теперь, поскольку вы знаете, как пользоваться командой **D** (Display), можно устанавливать адрес любой ячейки памяти для просмотра содержимого.

Можно также пролистывать память, периодически нажимая клавишу **D**, — DEBUG выведет на экран адреса, следующие за последней командой.

Для окончания работы и выхода из отладчика в DOS введите команду **Q** (Quit). Рассмотрим теперь использование отладчика DEBUG для непосредственного ввода программ в память и трассировки их выполнения.

Машинные команды имеют различную длину: один, два или три байта. Машинные команды находятся в памяти непосредственно друг за другом. Выполнение программы начинается с первой команды и далее последовательно выполняются остальные.

Можно ввести программу непосредственно в память машины и выполнить ее покомандно. В тоже время можно просматривать содержимое регистров после выполнения каждой команды. После загрузки DEBUG на экране высвечивается приглашение к вводу команд в виде дефиса.

Для непосредственного ввода программы на машинном языке введите следующую команду, включая пробелы:

```
E CS:100 B8 23 01 05 25 00 (нажмите Enter)
```

Команда **E** обозначает **Enter** (ввод). CS:100 определяет адрес памяти, куда будут вводиться команды, — шест.100 (256) байт от начала сегмента кодов. (Обычный стартовый адрес для машинных кодов в отладчике DEBUG).

Команда **E** записывает каждую пару шестнадцатеричных цифр в память в виде байта, начиная с адреса CS:100 до адреса CS:105.

Следующая команда **Enter**:

```
E CS:106 8B D8 03 D8 8B CB (Enter)
```

вводит шесть байтов в ячейки, начиная с адреса CS:106 и далее в 107, 108, 109, 10A и 10B. Последняя команда **Enter**:

```
E CS:10C 2B C8 2B C0 90 CB (Enter)
```

вводит шесть байтов, начиная с CS:10C в 10D, 10E, 10F, 110 и 111.

Проверьте правильность ввода значений. В случае, если есть ошибки, то следует повторить команды, которые были введены неправильно.

Теперь осталось самое простое — выполнить эти команды.

Введите команду R для просмотра содержимого регистров и флагов. В данный момент отладчик покажет содержимое регистров в шест. формате, например, AX=0000, BX=0000,...

Содержимое регистра IP (указатель команд) выводится в виде IP=0100, показывая что выполняемая команда находится на смещении 100 байт от начала сегмента кодов. Регистр флагов показывает следующие значения флагов:

```
NV UP DI PL NZ NA PO NC
```

Данные значения соответствуют:

- **NV —** нет переполнения
- **UP** — правое направление
- **DI** — прерывания запрещены
- **PL** — знак плюс
- **NZ** — не ноль
- **NA** — нет внешнего переноса
- **PO** — контроль на честность
- **NC** — нет переноса.

Команда **R** показывает также по смещению 0100 первую выполняемую машинную команду.

MOV AX,0123 — ассемблерный мнемонический код, соответствующий введенной машинной команде. Это есть результат операции дизассемблирования, которую обеспечивает отладчик для более простого понимания машинных команд. Рассматриваемая в данном случае команда обозначает пересылку непосредственного значения в регистр AX.

В данный момент команда **MOV** еще не выполнена. Для ее выполнения нажмите клавишу **T** (для трассировки) и клавишу **Enter**. В результате команда **MOV** будет выполнена и отладчик выдаст на экран содержимое регистров, флаги, а также следующую на очереди команду. Заметим, что регистр **AX** теперь содержит 0123. Машинная команда пересылки в регистр **AX** имеет код B8 и за этим кодом следует непосредственные данные 2301. В ходе выполнения команда B8 пересылает значение 23 в младшую часть регистра **AX**, то есть, однобайтовый регистр **AL**, а значение 01 — в старшую часть регистра AX, то есть, в регистр AH:

```
AX: | 01 | 23 |
```

Содержимое регистра IP:0103 показывает адрес следующей выполняемой команды в сегменте кодов:

```
13C6:0103 052500 ADD AX,0025
```

Для выполнения данной команды снова введите **T**. Команда прибавит 25 к младшей (**AL**) части регистра **AX** и 00 к старшей (**AH**) части регистра **AX**, то есть, прибавит 0025 к регистру **AX**. Теперь регистр **AX** содержит 0148, а регистр IP 0106 — адрес следующей команды для выполнения.

Введите снова команду **T**. Следующая машинная команда пересылает содержимое регистра **AX** в регистр **BX** и после ее выполнения в регистре **BX** будет содержаться значение 0148. Регистр **AX** сохраняет прежнее значение 0148, поскольку команда **MOV** только копирует данные из одного места в другое. Теперь вводите команду **T** для пошагового выполнения каждой оставшейся в программе команды. Следующая команда прибавит содержимое регистра **AX** к содержимому регистра **BX**, в последнем получим 0290. Затем программа скопирует содержимое регистра **BX** в **CX**, вычтет **AX** из **CX**, и вычтет **AX** из него самого. После этой последней команды, флаг нуля изменит свое состояние с **NZ** (не нуль) на **ZR** (нуль), так как результатом этой команды является нуль (вычитание **AX** из самого себя очищает этот регистр в 0).

Можно ввести **T** для выполнения последних команд **NOP** и **RET**, но это мы сделаем позже. Для просмотра программы в машинных кодах в сегменте кодов введите **D** для дампа:

```
D CS:100
```

В результате отладчик выдаст на каждую строку экрана по 16 байт данных в шест. представлении (32 шест. цифры) и в символьном представлении в коде ASCII (один символ на каждую пару шест. цифр). Представление машинного кода в символах ASCII не имеет смысла и может быть игнорировано.

Первая строка дампа начинается с 00 и представляет содержимое ячеек от CS:100 до CS:10F. Вторая строка представляет содержимое ячеек от CS:110 до CS:11F. Несмотря на то, что ваша программа заканчивается по адресу CS:111, команда **Dump** автоматически выдаст на восьми строках экрана дамп с адреса CS:100 до адреса CS:170.

При необходимости повторить выполнение этих команд сбросьте содержимое регистра **IP** и повторите трассировку снова. Введите **R IP**, введите 100, а затем необходимое число команд **T**. После каждой команды нажимайте клавишу **Enter**.

Для завершения работы с программой DEBUG введите **Q** (**Quit** — выход). В результате произойдет возврат в DOS и на экране появится приглашение **A>** или **C>**. В случае, если печатался протокол работы с отладчиком, то для прекращения печати снова нажмите **Ctrl/PrtSc**.



Определение данных

В предыдущем примере использовались непосредственные данные, описанные непосредственно в первых двух командах (**MOV** и **ADD**). Теперь рассмотрим аналогичный пример, в котором значения 0123 и 0025 определены в двух полях сегмента данных. Данный пример позволяет понять как компьютер обеспечивает доступ к данным посредством регистра **DS** и адресного смещения. В настоящем примере определены области данных, содержащие соответственно следующие значения:

Адрес в DS	Шест. знач.	Номера байтов
0000	2301	0 и 1
0002	2500	2 и 3
0004	0000	4 и 5
0006	2A2A2A	6, 7 и 8

Вспомним, что шест. символ занимает половину байта, таким образом, например, 23 находится в байте 0 (в первом байте) сегмента данных, 01 — в байте 1 (то есть, во втором байте).

Ниже показаны команды машинного языка, которые обрабатывают эти данные:

```
A10000
```

Переслать слово (два байта), начинающееся в DS по адресу 0000, в регистр AX.

```
03060200
```

Прибавить содержимое слова (двух байт), начинающегося в DS по адресу 0002, к регистру AX.

```
A30400
```

Переслать содержимое регистра AX в слово, начинающееся в DS по адресу 0004.

```
CB
```

Вернуться в DOS.

Обратите внимание, что здесь имеются две команды **MOV** с различными машинными кодами: **A1** и **A3**. Фактически машинный код зависит от регистров, на которые имеется ссылка, количества байтов (байт или слово), направления передачи данных (из регистра или в регистр) и от ссылки на непосредственные данные или на память.



Машинная адресация

Для доступа к машинной команде процессор определяет ее адрес из содержимого регистра CS плюс смещение в регистре IP. Например, предположим, что регистр CS содержит шест.04AF (действительный адрес 04AF0), а регистр IP содержит шест. 0023:

```
CS: 04AF0 IP: 0023
```

Адрес команды: 04B13

В случае, если, например, по адресу 04B13 находится команда:

```
A11200 MOV AX,[0012] | Адрес 04B13
```

то в памяти по адресу 04B13 содержится первый байт команды.

Процессор получает доступ к этому байту и по коду команды (A1) определяет длину команды — 3 байта.

Для доступа к данным по смещению [0012] процессор определяет адрес, исходя из содержимого регистра DS (как правило) плюс смещение в операнде команды. В случае, если DS содержит шест.04B1 (реальный адрес 04B10), то результирующий адрес данных определяется следующим образом:

```
DS: 04B10
```

Смещение: 0012

Адрес данных: 04B22

Предположим, что по адресам 04B22 и 04B23 содержатся следующие данные:

Содержимое: 24 01

Адрес: 04B22 04B23

Процессор выбирает значение 24 из ячейки по адресу 04B22 и помещает его в регистр **AL**, и значение 01 по адресу 04B23 — в регистр **AH**. Регистр **AX** будет содержать в результате 0124. В процессе выборки каждого байта команды процессор увеличивает значение регистра **IP** на единицу, так что к началу выполнения следующей команды в нашем примере **IP** будет содержать смещение 0026. Таким образом процессор теперь готов для выполнения следующей команды, которую он получает по адресу из регистра **CS** (04AF0) плюс текущее смещение в регистре **IP** (0026), то есть, 04B16.

Четная адресация

Процессоры действуют более эффективно, если в программе обеспечиваются доступ к словам, расположенным по четным адресам.

Процессор может сделать одну выборку слова по адресу 4B22 для загрузки его непосредственно в регистр. Но если слово начинается на нечетном адресе, процессор выполняет двойную выборку. Предположим, например, что команда должна выполнить выборку слова, начинающегося по адресу 04B23 и загрузить его в регистр **AX**:

Содержимое памяти: |xx|24|01|xx|

Адрес: 04B23

Сначала процессор получает доступ к байтам по адресам 4B22 и 4B23 и пересылает байт из ячейки 4B23 в регистр **AL**.

Затем он получает доступ к байтам по адресам 4B24 и 4B25 и пересылает байт из ячейки 4B23 в регистр **AH**. В результате регистр **AX** будет содержать 0124.

Нет необходимости в каких-либо специальных методах программирования для получения четной или нечетной адресации, не обязательно также знать является адрес четным или нет. Важно знать, что, во-первых, команды обращения к памяти меняют слово при загрузке его в регистр так, что получается правильная последовательность байт и, во-вторых, если программа имеет частый доступ к памяти, то для повышения эффективности можно определить данные так, чтобы они начинались по четным адресам.

Например, поскольку начало сегмента должно всегда находиться по четному адресу, первое поле данных начинается также по четному адресу и пока следующие поля определены как слова, имеющие четную длину, они все будут начинаться на четных адресах. В большинстве случаев, однако, невозможно заметить ускорения работы при четной адресации из-за очень высокой скорости работы процессоров.

Ассемблер имеет директиву **EVEN**, которая вызывает выравнивание данных и команд на четные адреса памяти.



Определение размера памяти

BIOS (базовая система ввода/вывода) в ROM имеет подпрограмму, которая определяет размер памяти. Можно обратиться в BIOS по команде **INT** по прерыванию 12H. В результате BIOS возвращает в регистр **AX** размер памяти в килобайтах.

Загрузите в память DEBUG и введите для **INT 12H** и **RET** следующие машинные коды:

```
E CS:100 CD 12 CB
```

Нажмите **R** (Enter) для отображения содержимого регистров и первой команды. Регистр **IP** содержит 0100, при этом высвечивается команда **INT 12H**.

Теперь нажмите **T** (и **Enter**) несколько раз и просмотрите выполняемые команды BIOS (отладчик показывает мнемокоды, хотя в дейст-

вительности выполняются машинные коды). В этот момент регистр **AX** содержит размер памяти в шестнадцатеричном формате. Теперь введите еще раз команду **T** для выхода из BIOS и возврата в вашу программу. На экране появится команда **RET** для машинного кода **CB**, который был введен вами.



Специальные средства отладчика

В операционной системе DOS можно использовать DEBUG для ввода команд Ассемблера так же, как и команд машинного языка. На практике можно пользоваться обоими методами.

Команда A

Команда отладчика A (Assemble) переводит DEBUG в режим приема команд Ассемблера и перевода их в машинные коды.

Команда U

Команда отладчика U (Unassemble) показывает машинные коды для команд Ассемблера. Необходимо сообщить отладчику адреса первой и последней команды, которые необходимо просмотреть

Ввод программы обычно используется на языке Ассемблера, когда машинный код неизвестен, а ввод в машинном коде — для изменения программы во время выполнения. Однако в действительности программа DEBUG предназначена для отладки программ.

Сохранение программы из отладчика

Можно использовать DEBUG для сохранения программ на диске в следующих случаях:

1. После загрузки программы в память машины и ее модификации необходимо сохранить измененный вариант. Для этого следует:

- загрузить программу по ее имени: **DEBUG n:имя файла** [Enter]
- просмотреть программу с помощью команды **D** и ввести изменения по команде **E**

- записать измененную программу: **W** [Enter]

2. Необходимо с помощью DEBUG написать небольшую по объему программу и сохранить ее на диске. Для этого следует:

- вызвать отладчик DEBUG
- с помощью команд **A** (assemble) и **E** (enter) написать программу
- присвоить программе имя: **N** имя файла.COM [Enter].

Тип программы должен быть COM — так как только программист знает, где действительно кончается его программа, указать отладчику длину программы в байтах.

- запросить регистр CX командой: R CX [Enter] — отладчик выдаст на этот запрос CX 0000 (нулевое значение)
- указать длину программы — 6
- записать измененную программу: **W** [Enter]

В обоих случаях DEBUG выдает сообщение «**Writing nnnn bytes**» (Запись nnnn байтов). В случае, если **nnnn** равно 0, то произошла ошибка при вводе длины программы, и необходимо повторить запись снова.

Важно: Отладчик DOS DEBUG это средство, полезное для отладки ассемблерных программ. Однако следует быть осторожным с ее использованием, особенно для команды **E** (ввод). Ввод данных в неправильные адреса памяти или ввод некорректных данных могут привести к непредсказуемым результатам.

На экране в этом случае могут появиться «странные» символы, клавиатура заблокирована или даже DOS превет DEBUG и перезагрузит себя с диска. Какие-либо серьезные повреждения вряд ли произойдут, но возможны некоторые неожиданности, а также потеря данных, которые вводились при работе с отладчиком.

В случае, если данные, введенные в сегмент данных или сегмент кодов, оказались некорректными, следует, вновь используя команду **E**, исправить их. Однако, можно не заметить ошибки и начать трассировку программы. Но и здесь возможно еще использовать команду **E** для изменений. В случае, если необходимо начать выполнение с первой команды, то следует установить в регистре командного указателя (IP) значение 0100.

Введите команду **R** (register) и требуемый регистр в следующем виде:

```
R IP [Enter]
```

Отладчик выдаст на экран содержимое регистра IP и перейдет в ожидание ввода. Здесь следует ввести значение 0100 и нажать для проверки результата команду **R** (без IP). Отладчик выдаст содержимое регистров, флагов и первую выполняемую команду. Теперь можно, используя команду **T**, вновь выполнить трассировку программы.

В случае, если ваша программа выполняет какие-либо подсчеты, то возможно потребуется очистка некоторых областей памяти и регистров. Но убедитесь в сохранении содержимого регистров CS, DS, SP и SS, которые имеют специфическое назначение.

Лекция 14. Макросредства



Простое макроопределение

Для каждой закодированной команды Ассемблер генерирует одну команду на машинном языке. Но для каждого закодированного оператора компиляторного языка Pascal или C генерируется один или более (чаще много) команд машинного языка. В этом отношении можно считать, что компиляторный язык состоит из макрооператоров.

Ассемблер MASM также имеет макросредства, но макросы здесь определяются программистом. Для этого задается имя макроса, директива MACRO, различные ассемблерные команды, которые должен генерировать данный макрос и для завершения макроопределения — директива MEND. Затем в любом месте программы, где необходимо выполнение определенных в макрокоманде команд, достаточно закодировать имя макроса. В результате Ассемблер сгенерирует необходимые команды.

Использование макрокоманд позволяет:

- упростить и сократить исходный текст программы;
- сделать программу более понятной;
- уменьшить число возможных ошибок кодирования.

Примерами макрокоманд могут быть операции ввода-вывода, связанные с инициализацией регистров и выполнения прерываний преобразования ASCII и двоичного форматов данных, арифметические операции над длинными полями, обработка строковых данных, деление с помощью вычитания.

Макроопределение должно находиться до определения сегмента.

Директива MACRO указывает Ассемблеру, что следующие команды до директивы ENDM являются частью макроопределения.

Директива ENDM завершает макроопределение. Команды между директивами MACRO и ENDM составляют тело макроопределения.

Имена, на которые имеются ссылки в макроопределении должны быть определены где-нибудь в другом месте программы.

Макрокоманда INIT1 может использоваться в кодовом сегменте там, где необходимо инициализировать регистры. Когда Ассемблер анализирует команду INIT1, он сначала просматривает таблицу мнемокодов и, не обнаружив там соответствующего элемента, проверяет макрокоманды. Так как программа содержит определение макрокоманды INIT1 Ассемблер подставляет тело макроопределения, генерируя необходимые команды — макрорасширение.

Программа использует рассматриваемую макрокоманду только один раз, хотя имеются другие макрокоманды, предназначенные на любое число применений и для таких макрокоманд Ассемблер генерирует одинаковые макрорасширения.



Использование параметров в макрокомандах

Формальные параметры в макроопределении указывают Ассемблеру на соответствие их имен любым аналогичным именам в теле макроопределения. Формальные параметры могут иметь любые правильные ассемблерные имена, не обязательно совпадающими именами в сегменте данных.

Формальный параметр может иметь любое правильное ассемблерное имя (включая имя регистра, например, CX), которое в процессе ассемблирования будет заменено на параметр макрокоманды. Отсюда следует, что Ассемблер не распознает регистровые имена и имена, определенные в области данных, как таковые. В одной макрокоманде может быть определено любое число формальных параметров, разделенных запятыми, вплоть до 120 колонки в строке.



Комментарии

Для пояснений назначения макроопределения в нем могут находиться комментарии. Директива COMMENT или символ точка с запятой указывают на строку комментария, как это показано в следующем макроопределении PROMPT:

```
PROMPT MACRO MESSGE ;Эта макрокоманда выводит сообщения на экран
MOV AH,09H
LEA DX,MESSGE
INT 21H
ENDM
```

Так как по умолчанию в листинг попадают только команды генерирующие объектный код, то Ассемблер не будет автоматически выдавать и комментарии, имеющиеся в макроопределении.

В случае, если необходимо, чтобы в расширении появлялись комментарии, следует использовать перед макрокомандой директиву .LALL («list all» — выводить все), которая кодируется вместе с лидирующей точкой:

```
.LALL PROMPT MESSAG1
```

Макроопределение может содержать несколько комментариев, причем некоторые из них могут выдаваться в листинге, а другие — нет. В первом случае необходимо использовать директиву .LALL. Во втором — кодировать перед комментарием два символа точка с запятой (;;) — признак подавления вывода комментария в листинг.

По умолчанию в Ассемблере действует директива .XALL, которая выводит в листинг только команды, генерирующие объектный код. И, наконец, можно запретить появление в листинге ассемблерного кода в макрорасширениях, особенно при использовании макрокоманды в одной программе несколько раз.

Для этого служит директива .SALL («suppress all» — подавить весь вывод), которая уменьшает размер выводимого листинга, но не оказывает никакого влияния на размер объектного модуля.

Директивы управления листингом .LALL, .XALL, .SALL сохраняют свое действие по всему тексту программы, пока другая директива листинга не изменит его. Эти директивы можно размещать в программе

так, чтобы в одних макрокомандах распечатывались комментарии, в других — макрорасширения, а в третьих подавлялся вывод в листинг.



Использование макрокоманд в макроопределениях

Макроопределение может содержать ссылку на другое макроопределение.

Рассмотрим простое макроопределение DOS21, которое заносит в регистр AH номер функции DOS и выполняет INT 21H:

```
DOS21 MACRO DOSFUNC
MOV AH,DOSFUNC
INT 21H
ENDM
```

Для использования данной макрокоманды при вводе с клавиатуры необходимо закодировать:

```
LEA DX,NAMEPAR DOS21 0AH
```

Предположим, что имеется другое макроопределение, использующее функцию 02 в регистре AH для вывода символа:

```
DISP MACRO CHAR
MOV AH,02
MOV DL,CHAR
INT 21H
ENDM
```

Для вывода на экран, например, звездочки достаточно закодировать макрокоманду DISP '*'. Можно изменить макроопределение DISP, воспользовавшись макрокомандой DOC21:

```
DISP MACRO CHAR
MOV DL,CHAR
DOS21 02
ENDM
```


Теперь, если закодировать макрокоманду DISP в виде DISP '*', то Ассемблер сгенерирует следующие команды:

```
MOV DL,'*'
MOV AH,02
INT 21H
```



Директива LOCAL

В некоторых макрокомандах требуется определять элементы данных или метки команд.

При использовании такой макрокоманды в программе более одного раза происходит также неоднократное определение одинаковых полей данных или меток. В результате Ассемблер выдаст сообщения об ошибке из-за дублирования имен.

Для обеспечения уникальности генерируемых в каждом макрорасширении имен используется директива LOCAL, которая кодируется непосредственно после директивы MACRO, даже перед комментариями. Общий формат имеет следующий вид:

```
LOCAL dummy-1,dummy-2,... ;Формальные параметры
```



Использование библиотек макроопределений

Определение таких макрокоманд, как INIT1 и INIT2 и одноразовое их использование в программе кажется бессмысленным. Лучшим подходом здесь является каталогизация собственных макрокоманд в библиотеке на магнитном диске, используя любое описательное имя, например, MACRO.LIB:

```
INIT MACRO CSNAME,DSNAME,SSNAME .
.
```

```
ENDM PROMPT MACRO MESSGE .
.
ENDM
```

Теперь для использования любой из каталогизированных макрокоманд вместо MACRO определения в начале программы следует применять директиву INCLUDE:

```
INCLUDE C:MACRO.LIB .
.
INIT CSEG,DATA,STACK
```

В этом случае Ассемблер обращается к файлу MACRO.LIB (в нашем примере) на дисководе С и включает в программу оба макроопределения INIT и PROMPT.

Хотя в нашем примере требуется только INIT. Ассемблерный листинг будет содержать копию макроопределения, отмеченного символом C в 30 колонке LST-файла.

Следом за макрокомандой идет ее расширение с объектным кодом и с символом плюс (+) в 31 колонке.

Так как транслятор с Ассемблера является двухпроходовым, то для обеспечения обработки директивы INCLUDE только в первом проходе (а не в обоих) можно использовать следующую конструкцию:

```
IF1 INCLUDE C:MACRO.LIB ENDIF
```

IF1 и ENDIF являются условными директивами. Директива IF1 указывает Ассемблеру на необходимость доступа к библиотеке только в первом проходе трансляции.

Директива ENDIF завершает IF-логику. Таким образом, копия макроопределений не появится в листинге — будет сэкономлено и время и память.

Директива очистки

Директива INCLUDE указывает Ассемблеру на включение всех макроопределений из специфицированной библиотеки.

Например, библиотека содержит макросы INIT, PROMPT и DIVIDE, хотя программе требуется только INIT. Директива PURGE позволяет «удалить» нежелательные макросы PROMPT и DIVIDE в текущем ассемблировании:

```
IF1 INCLUDE MACRO.LIB ;Включить всю библиотеку
ENDIF PURGE PROMRT,DIYIDE ;Удалить ненужные макросы ...
INIT CSEG,DATA,STACK ;Использование оставшейся макрокоманды
```


Директива PURGE действует только в процессе ассемблирования и не оказывает никакого влияния на макрокоманды, находящиеся в библиотеке.



Конкатенация (&)

Символ амперсанд (&) указывает Ассемблеру на сцепление (конкатенацию) текста или символов. Следующая макрокоманда MOVE генерирует команду MOVSB или MOVSW:

```
MOVE MACRO TAG REP MOVS&TAG ENDM
```

Теперь можно кодировать макрокоманду в виде MOVE B или MOVE W. В результате макрорасширения Ассемблер сцепит параметр с командой MOVS и получит REP MOVSB или REP MOVSW. Данный пример весьма тривиален и служит лишь для иллюстрации.



Директивы повторения: REPT, IRP, IRPC

Директивы повторения заставляют Ассемблер повторить блок операторов, завершаемых директивой ENDM.

Эти директивы не обязательно должны находится в макроопределении, но если они там находятся, то одна директива ENDM требуется для завершения повторяющегося блока, а вторая ENDM — для завершения макроопределения.

REPT: Повторение

Операция REPT приводит к повторению блока операторов до директивы ENDM в соответствии с числом повторений, указанным в выражении:

```
REPT выражение
```

В следующем примере происходит начальная инициализация значения N=0 и затем повторяется генерация DB N пять раз:

```
N = 0 REPT 5 N = N + 1 DB N ENDM
```

В результате будут сгенерированы пять операторов DB от DB 1 до DB 5.

Директива REPT может использоваться таким образом для определения таблицы или части таблицы. Другим примером может служить генерация пяти команд MOVSB, что эквивалентно REP MOVSB при содержимом CX равном 05:

```
REPT 5 MOVSB ENDM
```

IRP: Неопределенное повторение

Операция IRP приводит к повторению блока команд до директивы ENDM.

Основной формат:

```
IRP dummy,<arguments>
```

Аргументы, содержащиеся в угловых скобках, представляют собой любое число правильных символов, строк, числовых или арифметических констант.

Ассемблер генерирует блок кода для каждого аргумента. В следующем примере Ассемблер генерирует DB 3, DB 9, DB 17, DB 25 и DB 28:

```
IRP N,<3, 9, 17, 25, 28> DB N
ENDM
```

IRPC: Неопределенное повторение символа

Операция IRPC приводит к повторению блока операторов до директивы ENDM. Основной формат:

```
IRPC dummy,string
```

Ассемблер генерирует блок кода для каждого символа в строке «string». В следующем примере Ассемблер генерирует DW 3, DW 4 ... DW 8:

```
IRPC N,345678 DW N
ENDM
```




Условные директивы

Ассемблер поддерживает ряд условных директив. Условные директивы наиболее полезны внутри макроопределений, но не ограничены только этим применением.

Каждая директива IF должна иметь спаренную с ней директиву ENDIF для завершения IF-логики и возможную директиву ELSE для альтернативного действия.

Отсутствие директивы ENDIF вызывает сообщение об ошибке: «Undeterminated conditional» (незавершенный условный блок).

В случае, если проверяемое условие истинно, то Ассемблер выполняет условный блок до директивы ELSE или при отсутствии ELSE — до директивы ENDIF. В случае, если условие ложно, то Ассемблер выполняет условный блок после директивы ELSE, а при отсутствии ELSE вообще обходит условный блок.

Ниже перечислены различные условные директивы:

IF выражение

В случае, если выражение не равно нулю, Ассемблер обрабатывает операторы в условном блоке.

IFE выражение

В случае, если выражение равно нулю, Ассемблер обрабатывает операторы в условном блоке.

IF1 (нет выражения)

В случае, если осуществляется первый проход ассемблирования то обрабатываются операторы в условном блоке.

IF2 (нет выражения)

В случае, если осуществляется второй проход операторы ассемблирования, то обрабатываются в условном блоке.

IFDEF идентификатор

В случае, если идентификатор определен в программе или объявлен как EXTRN, то Ассемблер обрабатывает операторы в условном блоке.

IFNDEF идентификатор

В случае, если идентификатор не определен в программе или не объявлен как EXTRN, то Ассемблер обрабатывает операторы в условном блоке.

IFB <аргумент>

В случае, если аргументом является пробел, Ассемблер обрабатывает операторы в условном блоке. Аргумент должен быть в угловых скобках.

IFNB <аргумент>

В случае, если аргументом является не пробел, то Ассемблер обрабатывает операторы в условном блоке. Аргумент должен быть в угловых скобках.

IFIDN <арг-1>,<арг-2>

В случае, если строка первого аргумента идентична строке второго аргумента, то Ассемблер обрабатывает операторы в условном блоке. Аргументы должны быть в угловых скобках.

IFDIF<арг-1>,<арг-2>

В случае, если строка первого аргумента отличается от строки второго аргумента, то Ассемблер обрабатывает операторы в условном блоке.

Аргументы должны быть в угловых скобках.



Директива выхода из макроса EXITM

Макроопределение может содержать условные директивы, которые проверяют важные условия. В случае, если условие истинно, то Ассемблер должен прекратить дальнейшее макрорасширение. Для этой цели служит директива EXITM:

```
IFxx [условие] .
```



```
. (неправильное условие) .
EXITM .
.
ENDIF
```

Как только Ассемблер попадает в процессе генерации макрорасширения на директиву EXITM, дальнейшее расширение прекращается и обработка продолжается после директивы ENDM.

Можно использовать EXITM для прекращения повторений по директивам REPT, IRP и IRPC даже если они находятся внутри макроопределения.



Макрокоманды, использующие IF и IFNDEF

Макроопределение DIVIDE генерирует подпрограмму для выполнения деления вычитанием. Макрокоманда должна кодироваться с параметрами в следующей последовательности: делимое, делитель, частное.

Макрокоманда содержит директиву IFNDEF для проверки наличия параметров. Для любого неопределенного элемента макрокоманда увеличивает счетчик CNTR. Этот счетчик может иметь любое корректное имя и предназначен для временного использования в макроопределении. После проверки всех трех параметров, макрокоманда проверяет CNTR:

```
IF CNTR ;Макрорасширение прекращено
EXITM
```

В случае, если счетчик CNTR содержит ненулевое значение, то Ассемблер генерирует комментарий и прекращает по директиве EXITM дальнейшее макрорасширение. Заметим, что начальная команда устанавливает в счетчике CNTR нулевое значение и, кроме того, блоки IFNDEF могут устанавливать в CNTR единичное значение, а не увеличивать его на 1.

В случае, если Ассемблер успешно проходит все проверки, то он генерирует макрорасширение. В кодовом сегменте первая макрокоманда DIVIDE содержит правильные делимое и частное и, поэтому генерирует только комментарии.

Один из способов улучшения рассматриваемой макрокоманды — обеспечить проверку на ненулевой делитель и на одинаковый знак делимого и делителя; для этих целей лучше использовать коды Ассемблера, чем условные директивы.



Макрос, использующий IFIDN-условие

Макроопределение по имени MOVIF генерирует команды MOVSB или MOVSW в зависимости от указанного параметра. Макрокоманду можно кодировать с параметром B (для байта) или W (для слова) для генерации команд MOVSB или MOVSW из MOVS. Обратите внимание на первые два оператора в макроопределении:

```
MOVIF MACRO TAG
IFIDN <&TAG>,<B>
```

Условная директива IFIDN сравнивает заданный параметр (предположительно B или W) со строкой B. В случае, если значения идентичны, то Ассемблер генерирует REP MOVSB.

Обычное использование амперсанда (&) — для конкатенации, но в данном примере операнд <TAG> без амперсанда не будет работать. В случае, если в макрокоманде не будет указан параметр B или W, то Ассемблер сгенерирует предупреждающий комментарий и команду MOVSB (по умолчанию).

Примеры в кодовом сегменте трижды проверяют макрокоманду MOVIF: для параметра B, для параметра W и для неправильного параметра.

Не следует делать попыток выполнения данной программы в том виде, как она приведена на рисунке, так как регистры CX и DX не обеспечены правильными значениями.

Важно:

- Макросредства возможны только для полной версии Ассемблера (MASM).
- Использование макрокоманд в программах на Ассемблере дает в результате более удобочитаемые программы и более производительный код.

- Макроопределение состоит из директивы MACRO, блока из одного или нескольких операторов, которые генерируются при макрорасширениях и директивы ENDM для завершения определения.
- Код, который генерируется в программе по макрокоманде, представляет собой макрорасширение.
- Директивы .SALL, .LALL и .XALL позволяют управлять распечаткой комментариев и генерируемого объектного кода в макрорасширении.
- Директива LOCAL позволяет использовать имена внутри макроопределений. Директива LOCAL кодируется непосредственно после директивы MACRO.
- Использование формальных параметров в макроопределении позволяет кодировать параметры, обеспечивающие большую гибкость макросредств.
- Библиотека макроопределений дает возможность использовать макрокоманды для различных ассемблерных программ.
- Условные директивы позволяют контролировать параметры макрокоманд.

Лекция 15. Макропроцессоры



Основные понятия

Макропроцессор — модуль системного ПО, позволяющий расширить возможности языка Ассемблера за счет предварительной обработки исходного текста программы.

Определение, которое показанное выше, не представляется удачным, так как оно говорит только о сокращении объема записи, а это лишь одна из возможностей обеспечиваемых Макропроцессором. Хотя Макропроцессоры являются обязательным элементом всех современных языков Ассемблеров, аналогичные модули (Препроцессоры) могут быть и для других языков, в том числе и для языков высокого уровня. Для одних языков (Pascal, PL/1) применение средств препроцессора является опционным, для других (C, C++) — обязательным.

Важно понимать, что Макропроцессор осуществляет обработку исходного текста. Он «не вникает» в синтаксис и семантику операторов и переменных языка Ассемблера, не знает (как правило) имен, употребляемых в программе, а выполняет только текстовые подстановки. В свою очередь, Ассемблер обрабатывает исходный текст, не зная, написан тот или иной оператор программистом «своей рукой» или сгенерирован Макропроцессором. По тому, насколько Препроцессор (Макропроцессор) и Транслятор (Ассемблер) «знают» о существовании друг друга, их можно разделить на три категории:

- **Независимые.** Препроцессор составляет отдельный программный модуль (независимую программу), выполняющую просмотр (один или несколько) исходного модуля и формирующую новый файл исходного модуля, поступающий на вход Транслятора (пример — язык C).

- **Слабосвязанные.** Препроцессор составляет с Транслятором одну программу, но разные секции этой программы. Если в предыдущем случае Препроцессор обрабатывает весь файл, а затем передает его Транслятору, то в этом случае единицей обработки является каждый оператор исходного текста: он обрабатывается секцией Препроцессора, а затем передается секции Транслятора. (Пример — HLASM для S/390).
- **Сильносвязанные.** То же распределение работы, что и в предыдущем случае, но Препроцессор использует некоторые общие с Транслятором структуры данных. Например, Макропроцессор может распознавать имена, определенные в программе директивой EQU и т.п. (Пример — MASM, TASM).

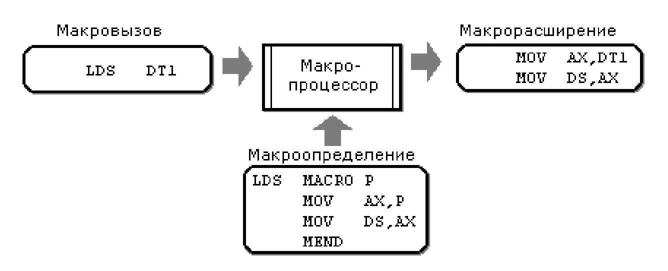
Основные термины, связанные с данными, обрабатываемыми Макропроцессором:

- макровызов (или макрокоманда);
- макроопределение;
- макрорасширение.

Макровызов или макрокоманда или макрос — оператор программы, который подлежит обработке Макропроцессором (Макропроцессор обрабатывает не все операторы, а только ему адресованные).

Макроопределение — описание того, как должна обрабатываться макрокоманда, макроопределение может находиться в том же исходном модуле, что и макрокоманда или в библиотеке макроопределений.

Макрорасширение — результат выполнения макровызова, представляющий собой один или несколько операторов языка Ассемблера, подставляемых в исходный модуль вместо оператора макровызова. Пример обработки макровызова показан на рисунке.



Оператор макровызова в исходной программе имеет тот же формат, что и другие операторы языка Ассемблера: В нем есть метка (необязательно), мнемоника и операнды. При обработке исходного текста если мнемоника оператора не распознается как машинная команда или директива, она считается макрокомандой и передается для обработки Макропроцессору.

Макроопределение описывает, как должна обрабатываться макрокоманда. Средства такого описания составляют некоторый Макроязык. Для Макропроцессоров 1-й и 2-й категорий средства Макроязыка могут быть достаточно развитыми. Для Макропроцессоров 3-й категории средства Макроязыка могут быть довольно бедными, но в составе языка Ассемблера может быть много директив, применяемых в макроопределениях (возможно, — только в макроопределениях). В теле макроопределения могут употребляться операторы двух типов:

- операторы Макроязыка, которые не приводят к непосредственной генерации операторов макрорасширения, а только управляют ходом обработки макроопределения;
- операторы языка Ассемблера (машинные команды и директивы), которые переходят в макрорасширение, возможно, с выполнением некоторых текстовых подстановок.

Поскольку макроопределение, обрабатывается перед трансляцией или вместе с ней, макрокоманда, определенная в исходном модуле, может употребляться только в этом исходном модуле и «не видна» из других исходных модулей. Для повторно используемых макроопределений обычно создаются библиотеки макроопределений. В некоторых системах (например, z/OS) макрокоманды обеспечивают системные вызовы и существуют богатейшие библиотеки системных макроопределений.

Самое очевидное применение макрокоманд — для сокращения записи исходной программы, когда один оператор макровызова заменяется на макрорасширение из двух и более операторов программы. В некоторых случаях макрорасширение может даже содержать и единственный оператор, но просто давать действию, выполняемому этим оператором более понятную мнемонику. Но возможности Макропроцессора гораздо шире. Так, одна и та же макрокоманда с разными параметрами может приводить к генерации совершенно различных макрорасширений — и по объему, и по содержанию.



Сравнение макросредств и подпрограмм

Использование макросредств во многом подобно использованию подпрограмм: в обоих случаях мы сокращаем запись исходного текста и создаем повторно используемые фрагменты кода. (Например, в C/C++ вызов псевдофункции неотличим от вызова функции.)

Принципиальные различия между подпрограммами и макросредствами:

- Команды, реализующие подпрограмму, содержатся в кода загрузочного модуля один раз, а команды, реализующие макровызов, включаются в программу для каждого применения макровызова (макросредства требуют больше памяти).
- Выполнение подпрограммы требует передачи управления с возвратом — команды типа CALL и RET, а команды макрорасширения включаются в общую последовательность команд программы (макровызовы выполняются быстрее).

Если в многофункциональной подпрограмме имеется разветвление в зависимости от значений параметров, то в загрузочный модуль включается код подпрограммы в полном объеме, даже если в конкретной программе реально используется только одна из ветвей алгоритма; в макровызове в каждое макрорасширение включаются только операторы, определяемые фактическими значениями параметров макровызова (экономия и времени и объема в макровызовах).

Общий итог сравнения: макросредства обеспечивают несколько большее быстродействие при несколько больших затратах памяти. Поэтому обычно макросредства применяются для оформления сравнительно небольших фрагментов повторяющегося кода.



Некоторые возможности Макроязыка

Ниже мы описываем некоторые возможности макроязыка, в той или иной форме реализованные во всех Макропроцессорах. Мы, однако, ориентируемся прежде всего на Макропроцессор, независимый от Ассемблера, потому что в этой категории функции Макропроцессора легче определить.

Заголовок макроопределения

Макроопределение должно как-то выделяться в программе, поэтому оно всегда начинается с заголовка.

Заголовок имеет формат, подобный следующему:

```
имя_макрокоманды MACRO список формальных параметров
```

имя_макрокоманды является обязательным компонентом. При макровызове это имя употребляется в поле мнемоники оператора. Имена макроопределений, имеющихся в программе, должны быть уникальны. Обычно при распознавании макровызова поиск по имени макрокоманды ведется сначала среди макроопределений имеющихся в программе, а затем (если в программе такое макроопределение не найдено) — в библиотеках макроопределений. Таким образом, имя макрокоманды, определенной в программе, может совпадать с именем макрокоманды, определенной в библиотеке, в этом случае макрокоманда, определенная в программе, заменяет собой библиотечную.

Формальные параметры играют ту же роль, что и формальные параметры процедур/функций. При обработке макровызова вместо имен формальных параметров в теле макроопределения подставляются значения фактических параметров макровызова.

В развитых Макроязыках возможны три формы задания параметров:

- позиционная;
- ключевая;
- смешанная.

При использовании позиционной формы соответствие фактических параметров формальным определяется их порядковым номером. (Позиционная форма всегда применяется для подпрограмм).

В позиционной форме количество и порядок следования фактических параметров макровызова должны соответствовать списку формальных параметров в заголовке макроопределения. При использовании ключевой формы каждый фактический параметр макровызова задается в виде:

```
имя_параметра=значение_параметра
```

В таком же виде они описываются и в списке формальных параметров, но здесь **значение_параметра** может опускаться. Если **значение_параметра** в списке формальных параметров не опущено, то это — значение по умолчанию. В макровызове параметры могут задаваться в любом порядке, параметры, имеющие значения по умолчанию, могут опускаться.

В смешанной форме первые несколько параметров подчиняются правилам позиционной формы, а остальные — ключевые.

В некоторых Макропроцессорах имена параметров начинаются с некоторого отличительного признака (например, амперсанда — &), чтобы Макропроцессор мог отличить «свои» имена (имена, подлежащие обработке при обработке макроопределения) от имен, подлежащих обработке Ассемблером. Для Макропроцессоров, которые мы отнесли к категории сильносвязанных такой признак может и не быть необходимым, так как такой Макропроцессор обрабатывает как свои имена, так и имена Ассемблера. В любом случае возникает проблема распознавания имени в теле макроопределения. Например, если макроопределение имеет формальный параметр &P, а в макровызове указано для него фактическое значение 'X', то как должна обрабатываться подстрока '&PA' в теле макроопределения? Должна ли эта подстрока быть заменена на 'XA' или оставлена без изменений?

Логика, которой следует большинство Макропроцессоров в этом вопросе, такова. &PA является именем в соответствии с правилами формирования имен. Поэтому оно не распознается как имя &P и остается без изменений. Если мы хотим, чтобы подстановка в этой подстроке все-таки произошла, следует поставить признак, отделяющий имя параметра от остальной части строки. Обычно в качестве такого признака используется точка — '.': '&P.A' заменяется на 'XA'.

Окончание макроопределения

Если у макроопределения есть начало (оператор MACRO), то у него, естественно, должен быть и конец. Конец макроопределения опреде-

ляется оператором MEND. Этот оператор не требует параметров. Макроопределение, взятое в «скобки» MACRO — MEND может располагаться в любом месте исходного модуля, но обычно все макроопределения размещают в начале или в конце модуля.



Локальные переменные макроопределения

Поскольку генерация макрорасширения ведется по некоторому алгоритму, описанному в макроопределении, реализация этого алгоритма может потребовать собственных переменных. Эти переменные имеют силу только внутри данного макроопределения, в макрорасширении не остается никаких «следов» переменных макроопределения.

Переменные макроопределения могут использоваться двумя способами:

- их значения могут подставляться вместо их имен в тех операторах макроопределения, которые переходят в макрорасширение;
- их значения могут проверяться в условных операторах макроязыка и влиять на последовательность обработки.

При подстановке значений переменных макроопределения в макрорасширение работают те же правила, что и при подстановки значений параметров.

Для сильносвязанных Макропроцессоров необходимости в локальных переменных макроопределения, вместо них могут использоваться имена программы (определяемые директивой EQU). Для сильносвязанных и независимых процессоров переменный макроопределения и имена программы должны различаться, для этого может применяться тот же признак, что и для параметров макроопределения.

Объявление локальной переменной макроопределения может иметь, например, вид:

```
имя_переменной LOCL начальное_значение (последнее необязательно)
```




Присваивание значений переменным макроопределения

Присваивание может производиться оператором вида:

```
имя_переменной SET выражение
```

или

```
имя_переменной = выражение
```

Выражения, допустимые при присваивании, могут включать в себя имена переменных и параметров макроопределения, константы, строковые, арифметические и логические операции, функции.

Основной тип операций — строковые (выделение подстроки, поиск вхождения, конкатенация. etc.), так как обработка макроопределения состоит в текстовых подстановках.

Строковые операции обычно реализуются в функциях. Однако, в некоторых случаях может потребоваться выполнение над переменными макроопределения операций нестрокового типа.

Как обеспечить выполнение таких операций? Можно предложить два варианта решения этой проблемы:

- Ввести в оператор объявления переменной макроопределения определение ее типа. При выполнении операций должно проверяться соответствие типов.
- Все переменные макроопределения имеют строковый тип, но при вычислении выражений автоматически преобразуются к типу, требуемому для данной операции (при таком преобразовании может возникать ошибка). Результат выражения автоматически преобразуется в строку.

Как правило, операции присваивания могут применяться к параметрам макроопределения точно так же, как и к переменным макроопределения.



Глобальные переменные макроопределения

Значения локальных переменных макроопределения сохраняются только при обработке данного конкретного макровызова. В некоторых случаях, однако, возникает необходимость, чтобы значение переменной макроопределения было запомнено Макропроцессором и использовано при следующей появлении той же макрокоманды в данном модуле. Для этого могут быть введены глобальные переменные макроопределения (в сильносвязанных Макропроцессорах в них опять-таки нет необходимости).

Объявление глобальной переменной макроопределения может иметь, например, вид:

```
имя_переменной GLBL начальное_значение (последнее необязательно)
```

Присваивание значений глобальным переменным макроопределения выполняется так же, как и локальным.



Уникальные метки

В некоторых случаях операторы машинных команд, имеющихся в макроопределении, должны быть помечены, например, для того, чтобы передавать на них управление. Если применить для этих целей обычную метку, то может возникнуть ошибочная ситуация. Если метка в макроопределении имеет обычное имя, и в модуле данная макрокоманда вызывается два раза, то будет сгенерировано два макрорасширения, и в обоих будет метка с этим именем. Чтобы избежать ситуации неуникальности меток, в макроязыке создается возможность определять метки, для которых формируются уникальные имена. Обычно имя такой метки имеет тот же отличительный признак, который имеют параметры и переменные макроопределения. Каждую такую метку Макропроцессор заменяет меткой с уникальными именем.

Уникальное имя метки может формироваться формате, подобном следующему:

```
&имя.nnnnnn
```

где — **nnnnnn** — число, увеличивающееся на 1 для каждой следующей уникальной метки.

Другой возможный способ формирования, например:

```
имя&SYSNDX
```

где **SYSNDX** — предустановленное имя, имеющее числовое значение, начинающееся с 00001 и увеличивающееся на 1 для каждой следующей уникальной метки.

Следующие операторы Макроязыка влияют на последовательность обработки операторов макроопределения. В тех или иных Макропроцессорах имеется тот или иной набор таких операторов.

Оператор безусловного перехода и метки макроопределения

Возможный формат оператора:

```
MGO макрометка
```

Концептуально важным понятием является макрометка. Макрометка может стоять перед оператором Макроязыка или перед оператором языка Ассемблера. Макрометки не имеют ничего общего с метками в программе. Передача управления на макрометку означает то, что при обработке макроопределения следующим будет обрабатываться оператор, помеченный макрометкой. Макрометки должны иметь какой-то признак, по которому их имена отличались бы от имен программы и переменных макроопределения. Например, если имена переменных макроопределения начинаются с символа &, то имя макрометки может начинаться с &&.

Оператор условного перехода

Возможный формат оператора:

```
MIF условное_выражение макрометка
```

Если условное_выражение имеет значение «истина», обработка переходит на оператор, помеченный макрометкой, иначе обрабатывается следующий оператор макроопределения. Условные выражения формируются по обычным правилам языков программирования. В них могут употребляться параметры и переменные (локальные и глобальные) макроопределения, константы, строковые, арифметические и логические операции и, конечно же, операции сравнения.

Кроме того, в составе Макроязыка обычно имеются специальные функции, позволяющие распознавать тип своих операндов, например:

является ли операнд строковым представлением числа, является ли операнд именем, является ли операнд именем регистра.

Условные блоки

Возможный формат оператора:

```
IF условное_выражение
операторы_макроопределения_блок1
ENDIF
ELSE
операторы_макроопределения_блок2
ENDIF
```

Если условное_выражение имеет значение «истина», обрабатываются операторы макроопределения от оператора IF до оператора ENDIF, иначе обрабатываются операторы макроопределения от оператора ESLE до оператора ENDIF. Как и в языках программирования блок ELSE — ENDIF не является обязательным.

Условные выражения описаны выше. Обычно предусматриваются специальные формы:

```
IFDEF имя
IFNDEF имя
```

проверяющие просто определено или не определено данное имя.

Операторы условных блоков довольно часто являются не операторами Макроязыка, а директивами самого языка Ассемблера.



Операторы повторений

Операторы повторений Макроязыка (или директивы повторений языка Ассемблера) заставляют повторить блок операторов исходного текста, возможно, с модификациями в каждом повторении. Операторы повторений играют роль операторов цикла в языках программирования, они не являются обязательными для макроязыка, так как цикл можно обеспечить и условным переходом.

Как и в языках программирования, в Макроязыке может быть несколько форм операторов повторения, приведем некоторые (не все) из возможных форм:


```
MDO выражение
блок_операторов_макроопределения
ENDMDO
```

выражение должно иметь числовой результат, обработка блока операторов повторяется столько раз, каков результат вычисления выражения.

```
MDOLIST переменная_макроопределения,
список_выражений
блок_операторов_макроопределения
ENDMDO
```

обработка блока операторов повторяется столько раз, сколько элементов имеется в списке_выражений, при этом в каждой итерации переменной_макроопределения присваивается значение очередного элемента из списка_выражений.

```
MDOWHILE условное_выражение
блок_операторов_макроопределения
ENDMDO
```

обработка блока операторов повторяется до тех пор, пока значение условного_выражения — «истина».



Выдача сообщения

При возникновении ошибок или ситуаций, требующих предупреждения программисту в листинг должно выводиться сообщение. Если в результате ошибки программиста, написавшего макроопределение или макровызов будет сгенерирован неправильный код программы на языке Ассемблера, то эта ошибка будет выявлена только Ассемблером на этапе трансляции программы. Однако выгоднее выявлять ошибки не как можно более ранних этапах подготовки программы, в Макроязыке ошибочные ситуации (ошибки в параметрах и т.п.) могут быть выявлены при помощи условных операторов или блоков, а для выдачи сообщения об ошибке должен существовать специальный оператор Макроязыка. Формат такого оператора примерно следующий:

```
MOTE код_серьезности,код_ошибки,сообщение_об_ошибке
```

код_серьезности — числовой код, определяющий возможность продолжения работы при наличии ситуации, вызвавшей сообщения.

Должны индицироваться, как минимум, следующие ситуации:

- работа Макропроцессора может быть продолжена, по окончании ее может выполняться ассемблирование;
- работа Макропроцессора может быть продолжена, но ассемблирование выполняться не может;
- работа Макропроцессора не может продолжаться.

код_ошибки — числовой код, служащий, например, для поиска развернутого описания сообщений и действий при его возникновении в документе «Сообщения программы»

сообщение_об_ошибке — текст, печатаемый в листинге



Завершение обработки

Обработка макроопределения завершается при достижении оператора MEND. Однако, поскольку алгоритм обработки макроопределения может разветвляться, должна быть предусмотрена возможность выхода из обработки и до достижения конца макроопределения. Эта возможность обеспечивается оператором MEXIT. Операндом этого оператора может быть код_серьезности.



Комментарии макроопределения

Если в тексте макроопределения имеются комментарии, то они переходят в макрорасширение так же, как и операторы машинных команд и директив Ассемблера. Однако, должна быть обеспечена и возможность употребления таких комментариев, которые не переходят в макрорасширение — комментарии, которые относятся не к самой программе, а к макроопределению и порядку его обработки. Такие комментарии должны обладать некоторым отличительным признаком. Возможны специальные директивы Ассемблера, определяющие режим печати комментариев макроопределения.



Макрорасширения в листинге

Как уже неоднократно говорилось, макрорасширения для Ассемблера неотличимы от программного текста, написанного программистом «своей рукой». Но программист, анализируя листинг программы, конечно, должен видеть макрорасширения и отличать их от основного текста. Как правило, директивы Ассемблера, управляющие печатью листинга предусматривают режим, при котором макрорасширение не печатается в листинге, а печатается только макрокоманда и режим, при котором в листинге печатается и макрокоманда, и ее макрорасширение, но операторы макрорасширения помечаются каким-либо специальным символом.

Структуры данных Макропроцессора

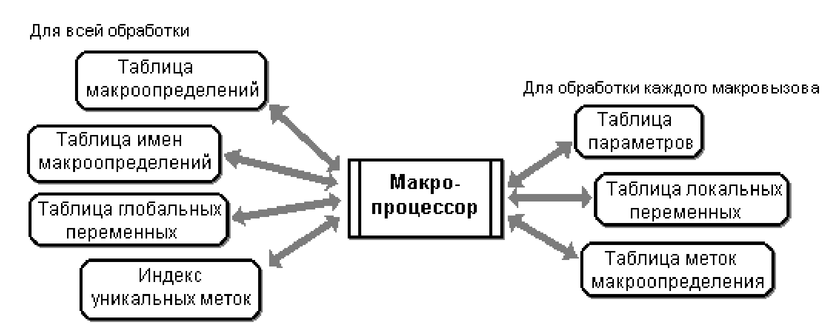
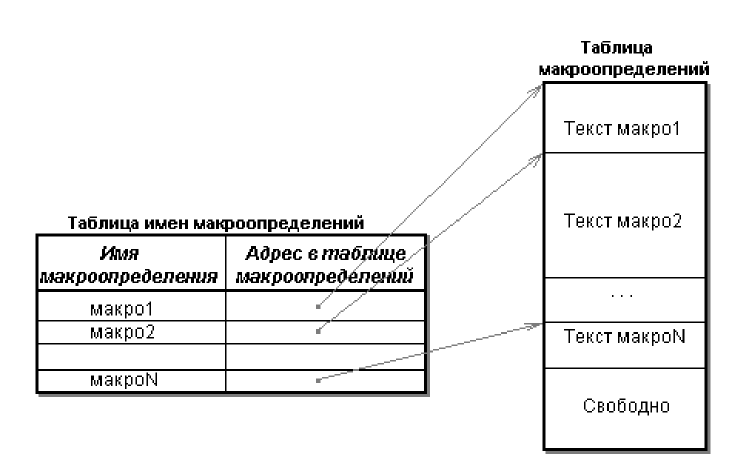


Таблица макроопределений, строго говоря, не таблица, а просто массив строк, в который записываются тексты всех макроопределений (от оператора MACRO до оператора MEND), найденных в обрабатываемом модуле.

Таблица имен макроопределений содержит имена макроопределений и указатель на размещение текста макроопределения в таблице макроопределений, как показано на рисунке.



Все таблицы имеют переменный размер и заполняются в процессе работы. Индекс уникальных меток — число, используемое для формирования уникальной части имен меток, встречающихся в макроопределениях

Для обработки каждого макровызова создаются:

- Таблица параметров, содержащая информацию о параметрах макроопределения.
- Таблица локальных переменных, содержащая информацию о локальных переменных макроопределения.

Структура этих таблиц — такая же, как и таблицы глобальных переменных, эти две таблицы могут быть объединены в одну таблицу параметров и локальных переменных.

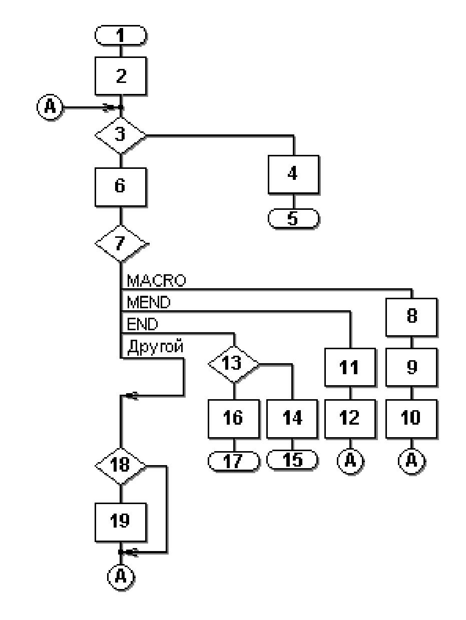


Алгоритм работы Макропроцессора

Очевидно, что когда Макропроцессор обрабатывает макровызов, он уже должен «знать» макроопределение данной макрокоманды. Для обеспечения этого таблицы макроопределений и имен макроопределений должны быть созданы до начала обработки макровызовов. Поэтому Макропроцессор должен состоять из двух проходов, на первом проходе строятся таблицы макроопределений и имен макроопределений, а на втором осуществляется обработка макровызовов. Если макроопределения сосредоточены в начале исходного модуля, то Макропроцессор может быть и однопроходным. Ниже мы приводим алгоритм работы 2-проходного Макропроцессора, при этом мы исходим из следующих предпосылок:

- наш Макропроцессор является независимым от Ассемблера;
- таблица параметров объединяется с таблицей локальных переменных, в дальнейшем мы называем объединенную таблицу таблицей локальных переменных;
- операторы Макроязыка включают в себя: MACRO, MEND, MEXIT, MNOTE, LOCL, GLBL, SET, MGO, MIF;
- обеспечиваются локальные и глобальные переменные макроопределений, уникальные метки.

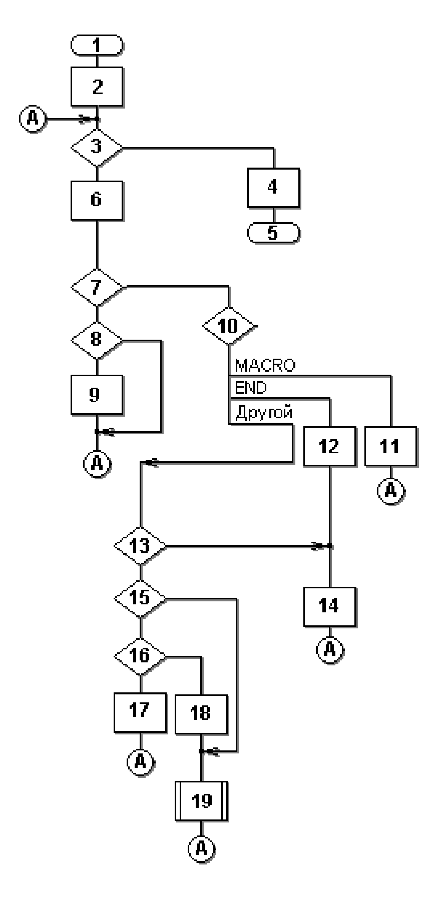
Алгоритм выполнения 1-го прохода следующий:



- **Блок1:** 1-й проход Макропроцессора
- **Блок2:** Инициализация: открытие исходного файла, создание пустых таблиц, признак «обработка макроопределения» устанавливается в FALSE.
- **Блок3:** Чтение следующей строки исходного файла с проверкой конца файла.
- **Блок4:** Если при чтении строки найден конец файла, выводится сообщение об ошибке, закрываются файлы, освобождается память...
- **Блок5:** ...и Макропроцессор завершается с признаком ошибки.
- **Блок6:** Если конец файла не достигнут, выполняется лексический разбор прочитанной строки с выделением имени и мнемоники операции.
- **Блок7:** Алгоритм Макропроцессора разветвляется в зависимости от мнемоники операции
- **Блок8:** Если мнемоника операции MACRO — заголовок макроопределения, то в таблицу имен макроопределений заносится имя, находящееся в этом операторе и начальный адрес свободной области в таблице макроопределений. (При занесении имени в таблицу имен макроопределений проверяется, нет ли уже в таблице такого имени, если есть — ошибка)
- **Блок9:** Оператор MACRO записывается в таблицу макроопределений.
- **Блок10:** Признак «обработка макроопределения» устанавливается в TRUE.
- **Блок11:** Если мнемоника операции MEND — конец макроопределения, то оператор записывается в таблицу макроопределений...
- **Блок12:** ...и признак «обработка макроопределения» устанавливается в FALSE.
- **Блок13:** Если мнемоника операции END — конец программы, то проверяется установка признака «обработка макроопределения».
- **Блок14:** Если этот признак установлен в TRUE, т.е., конец программы встретился до окончания макроопределения, то выводится сообщение об ошибке, закрываются файлы, освобождается память...

- **Блок15:** ...и Макропроцессор завершается с признаком ошибки.
- **Блок16:** Если этот признак установлен в FALSE, то выполняются завершающие операции ...
- **Блок17:** ...и заканчивается 1-й проход Макропроцессора.
- **Блок18:** При любой другой мнемонике оператора проверяется установка признака «обработка макроопределения».
- **Блок19:** Если этот признак установлен в TRUE, то оператор записывается в таблицу макроопределений, если признак установлен в FALSE, то оператор игнорируется Макропроцессором.

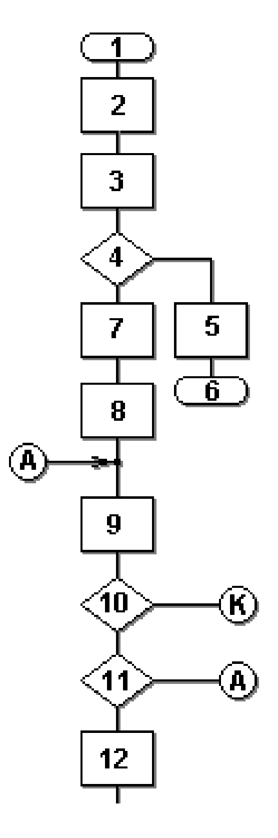
Алгоритм выполнения 2-го прохода следующий:

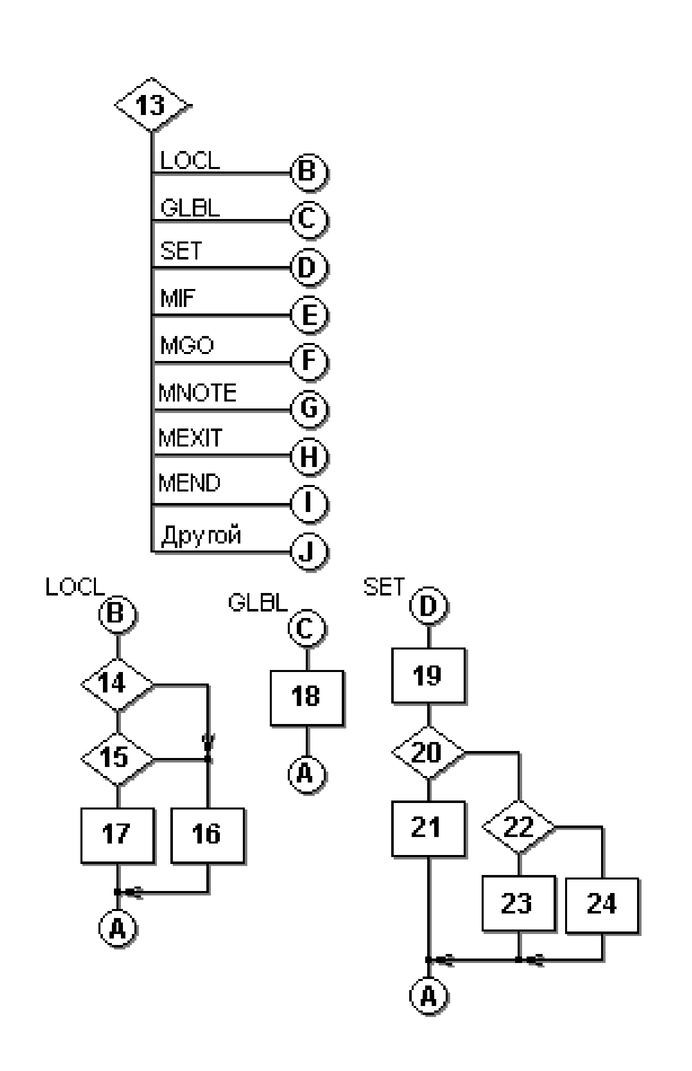


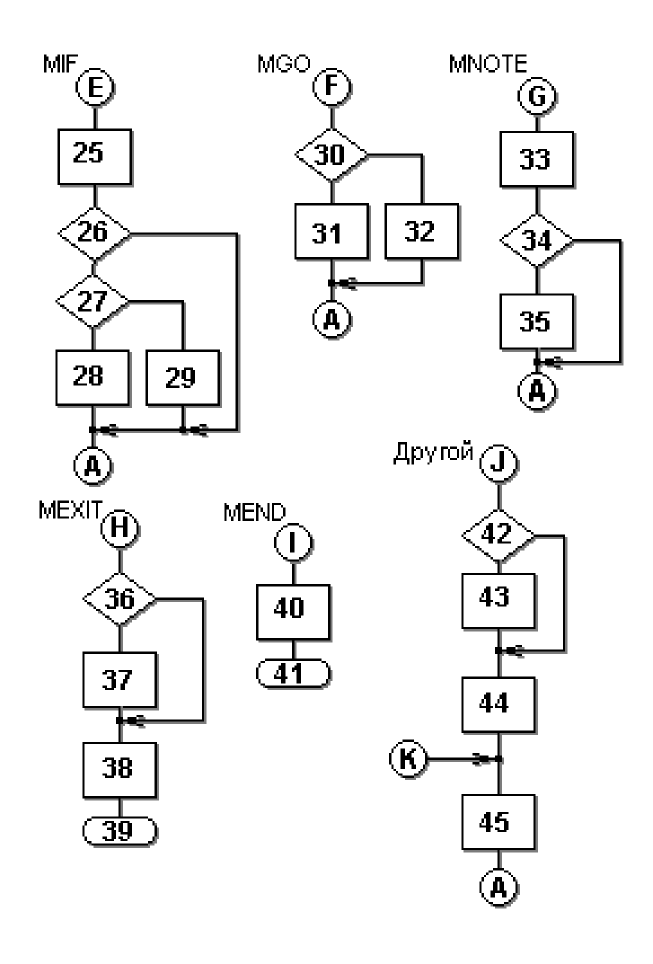
- **Блок1:** 2-й проход макропроцессора
- **Блок2:** Начальные установки: открытие файлов, создание пустых таблиц. Признак режима обработки устанавливается в значение «обработка программы».
- **Блок3:** Признак конца обработки установлен?
- **Блок4:** Если признак конца обработки установлен, выполняются завершающие операции...
- **Блок5:** ...и работа Макропроцессора заканчивается.
- **Блок6:** Выполняется разбор строки.
- **Блок7:** Проверяется признак режима обработки.
- **Блок8:** Если признак режима установлен в значение «обработка макроопределения», то проверяется мнемоника оператора.
- **Блок9:** Если в режиме обработки макроопределения встречается мнемоника MEND, то режим переключается в «обработка программы», все прочие операторы в режиме обработки макроопределения игнорируются.
- **Блок10:** Если признак режима работы установлен в значение «обработка программы», происходит ветвление алгоритма в зависимости от мнемоники оператора.
- **Блок11:** Обработка оператора MACRO заключается в установке режима обработки в значение «обработка программы».
- **Блок12:** Обработка директивы Ассемблера END заключается в установке признака окончания работы и выводе оператора в выходной файл.
- **Блок13:** Любая другая мнемоника ищется в Таблице машинных команд и в Таблице директив Ассемблера. Если мнемоника найдена в одной из этих таблиц, то...
- **Блок14:** ...оператор просто выводится в выходной файл.
- **Блок15:** Если оператор не является оператором языка Ассемблера, то предполагается, что это макровызов и соответствующее мнемонике имя ищется в Таблице имен макроопределений.
- **Блок16:** Если имя не найдено в Таблице имен макроопределений, то оно ищется в библиотеках макроопределений.

- **Блок17:** Если имя не найдено и в библиотеках макроопределений, вырабатывается сообщение об ошибке и управление передается на чтение следующего оператора программы.
- **Блок18:** Если имя не найдено в библиотеках макроопределений, соответствующие элементы включаются в Таблицу имен макроопределений и в Таблицу макроопределений.
- **Блок19:** Если имя есть в Таблице макроопределений, выполняется обработка макровызова, после чего управление передается на чтение следующего оператора программы.

Алгоритм обработки макровызова следующий:







- **Блок1:** Обработка макровызова. На входе этого модуля есть номер элемента в Таблице имен макроопределений и разобранный текст оператора макровызова.
- **Блок2:** Создание пустых: Таблицы локальных переменных, Таблицы меток.
- **Блок3:** Чтение первой строки из Таблицы макроопределений по адресу, записанному в элементе Таблице имен макроопределений. (Здесь и далее мы подразумеваем, что после чтения очеред-

ной строки макроопределения указатель для следующего чтения устанавливается на адрес следующей строки, если он не изменен явным образом.)

- **Блок4:** Проверка параметров: сопоставление фактических параметров вызова с формальными параметрами, описанными в заголовке макроопределения (Заголовок находится в строке, только что считанной из Таблицы макроопределений).
- **Блок5:** При несоответствии фактических параметров формальным выдается сообщение об ошибке...
- **Блок6:** ...и обработка макровызова завершается
- **Блок7:** При правильном задании фактических параметров параметры и их значения заносятся в Таблицу локальных переменных.
- **Блок8:** Создается и заполняется Таблица меток макроопределения. При этом текст макроопределения просматривается до оператора MEND, выявляются метки и заносятся в таблицу. Проверяется уникальность меток. После заполнения таблицы меток указатель чтения из Таблицы макроопределений устанавливается на вторую (следующую за заголовком строку) текста макроопределения.
- **Блок9:** Читается следующая строка текста макроопределения.
- **Блок10:** Если строка является комментарием Ассемблера, строка выводится в макрорасширение.
- **Блок11:** Если строка является комментарием Макроязыка, управление передается на чтение следующей строки макроопределения.
- **Блок12:** Выполняется разбор строки.
- **Блок13:** Алгоритм ветвится в зависимости от мнемоники оператора.
- **Блок14:** При обработке оператора LOCL имя локальной переменной ищется сначала в Таблице локальных переменных...
- **Блок15:** ...а затем — в Таблице глобальных переменных.
- **Блок16:** Если имя найдено в одной из таблиц, формируется сообщение о неуникальном имени.

- **Блок17:** В противном случае заносится новая строка в таблицу локальных имен. В любом случае управление передается на чтение следующей строки макроопределения.
- **Блок18:** Обработка оператора GLBL отличается от оператора LOCL только тем, что новая строка создается в Таблице глобальных переменных.
- **Блок19:** При обработке оператора LOCL вычисляется выражение — операнд команды. Вычисление включает в себя подстановку значений входящих в выражение переменных. Возможны ошибки — из-за использования неопределенных имен и ошибок в синтаксисе выражения.
- **Блок20:** Имя переменной ищется сначала в Таблице локальных переменных.
- **Блок21:** Если имя найдено, изменяется его значение в Таблице локальных переменных.
- **Блок22:** Если имя переменной не найдено, оно ищется в Таблице глобальных переменных.
- **Блок23:** Если имя найдено в Таблице глобальных переменных, изменяется его значение в этой таблице.
- **Блок24:** Если имя не найдено ни в одной из таблиц, формируется сообщение о неопределенном имени.
- **Блок25:** При обработке оператора MIF вычисляется условное выражение — 1-й операнд команды (возможны ошибки).
- **Блок26:** Проверяется значение вычисленного условного выражения.
- **Блок27:** Если значение выражения «истина», имя метки — 2-го операнда команды ищется в Таблице меток макроопределения.
- **Блок28:** Если метка найдена в таблице, указатель для следующего чтения из Таблице макроопределений устанавливается на адрес соответствующий метке
- **Блок29:** Если метка найдена в таблице, выдается сообщение о неопределенной метке.
- **Блок30:** При обработке оператора MGO имя метки — операнда команды ищется в Таблице меток макроопределения.

- **Блок31:** Если метка найдена в таблице, указатель для следующего чтения из Таблице макроопределений устанавливается на адрес соответствующий метке.
- **Блок32:** Если метка найдена в таблице, выдается сообщение о неопределенной метке.
- **Блок33:** При обработке оператора MNOTE выводится сообщение, определяемое операндом.
- **Блок34:** Устанавливается и анализируется код серьезности. Код серьезности является общим для всей работы Макропроцессора, его значение изменяется только, если новое значение больше текущего (более серьезная ошибка)
- **Блок35:** Если код серьезности не допускает продолжения работы Макропроцессора, устанавливается признак завершения работы.
- **Блок36:** При обработке оператора MEXIT устанавливается и анализируется код серьезности.
- **Блок37:** Если код серьезности не допускает продолжения работы Макропроцессора, устанавливается признак завершения работы.
- **Блок38:** Освобождаются структуры данных, созданные для обработки макровызова...
- **Блок39:** ...и обработка макровызова завершается.
- **Блок40:** При обработке оператора MEND освобождаются структуры данных, созданные для обработки макровызова...
- **Блок41:** ...и обработка макровызова завершается.
- **Блок42:** Любая другая мнемоника операции означает, что оператор является не оператором Макроязыка, а оператором языка Ассемблера. В этом случае прежде всего проверяется, не имеет ли оператор метки, которая должна быть уникальной.
- **Блок43:** Если оператор имеет такую метку, формируется имя уникальной метки и индекс уникальных меток увеличивается на 1.
- **Блок44:** Выполняются подстановки в операторе языка Ассемблера (значение имен ищутся в Таблицах локальных и глобальных переменных, возможны ошибки).
- **Блок45:** Оператор языка Ассемблера записывается в макрорасширение.





Библиотеки макроопределений

Макровызовы к макроопределение, приведенному в исходном модуле, могут применяться только в этом же исходном модуле. Для того, чтобы можно было использовать макроопределение в разных исходных модулях, макроопределения помещаются в библиотеку макроопределений. Список библиотек макроопределений, которые используются для данного исходного модуля является параметром Макропроцессора.

Мы в нашей схеме алгоритма показали, что обращение к библиотекам макроопределений происходит на 2-м проходе Макропроцессора — если мнемоника оператора не распознана ни как оператор языка Ассемблера, ни как макрокоманда, определенная в данном исходном модуле. Возможны, однако, и другие алгоритмы использования библиотек.

Один из таких алгоритмов следующий.

Анализ мнемоники производится на 1-м проходе Ассемблера, все операторы, не распознанные как операторы языка Ассемблера, считаются макрокомандами и для них создаются строки в Таблице имен макроопределений.

Если для такой макрокоманды макроопределение еще не найдено, поле ссылки на Таблицу макроопределений остается пустым.

Если в исходном модуле встречается макроопределение, то его текст заносится в Таблицу макроопределений. Если в Таблице имен макроопределений уже есть это имя с пустой ссылкой на Таблицу макроопределений, ссылке присваивается значение. Если такого имени в Таблице имен макроопределений нет, в таблице создается новая строка.

В конце 1-го прохода просматривается Таблица имен макроопределений. Если в таблице находятся имена с пустыми ссылками на Таблицу макроопределений, соответствующее макроопределение ищется в библиотеках. Если макроопределение найдено в библиотеке, его текст переписывается в Таблицу макроопределений и присваивается значение ссылке в соответствующей строке Таблицы имен макроопределений.

Если после этого в Таблице имен макроопределений остаются имена с пустыми ссылками, это свидетельствует об ошибках в программе.



Вложенные макровызовы. Вложенные макроопределения

Можно ли употреблять макроопределения внутри макроопределений? Можно ли употреблять макровызовы вызовы внутри макроопределений? Представленные выше алгоритмы делать этого не позволяют. Тем не менее, можно построить такие алгоритмы Макропроцессора, которые это позволять будут. Эти алгоритмы в любом случае будут довольно «затратными», то есть, требующими много ресурсов — процессорного времени и памяти. «Классические» алгоритмы, создававшиеся в условиях хронического дефицита памяти, были очень ограничены.

Макроопределения внутри макроопределений

Честно говоря, необходимость в таких средствах сомнительна. Она может возникнуть при создании большого макроопределения, в котором есть повторяющиеся фрагменты. Вложенное макроопределение действительно только внутри того макроопределения, в которое оно вложено.

Против такого средства можно привести 2 соображения:

- макроопределение не бывает слишком большим — иначе не срабатывают его преимущества над подпрограммой (следует однако признать, что могут существовать довольно большие макроопределения, которые генерируют разнообразные варианты небольших макрорасширений);
- в языке Pascal допускаются вложенные процедуры, а в языке C — нет; и C прекрасно обходится без них, да и современная практика программирования на Pascal их практически не использует.

Тем не менее, если вложенные макроопределения все же необходимы, можно предложить следующий вариант их реализации: 1-й проход Макропроцессора работает почти по тому же алгоритму, который приведен нами. Принципиально важно, однако, что Таблица макроопределений и Таблица имен макроопределений имеют последовательную структуру, элементы в них записываются в порядке их поступления.

В Макропроцессоре есть некоторая целая переменная — глубина вложенности. Ее исходное значение — 0, при каждом появлении оператора MACRO это значение увеличивается на 1, при каждом появлении оператора MEND — уменьшается на 1. Если при глубине вложенности 0 появляется оператор MACRO, в Таблицу имен макроопределений заносится новый элемент, и текст макроопределения записывается в Таблицу макроопределений — до тех пор, пока глубина вложенности не станет равной 0.

Появление оператора MACRO при глубине вложенности, большей 0 не приводит к созданию нового элемента в Таблице имен макроопределений.

Таким образом, в Таблице имен макроопределений имеется строка только для самого внешнего макроопределения, а все вложенные пока «не видны» и находятся внутри текста внешнего в Таблице макроопределений.

2-й проход Макропроцессора при обработке макровызова считывает текст макроопределения в некоторый буфер и прежде всего рекурсивно вызывает для его обработки Макропроцессор.

Для вложенного вызова Макропроцессора доступны Таблица макроопределений и Таблица имен макроопределений, новые макроопределения, обнаруженные рекурсивным вызовом заносятся в конец этих таблиц.

При возврате из рекурсивного вызова макроопределения, дописанные им, удаляются из таблиц.

Макрокоманды внутри макроопределений

В отличие от предыдущего, это средство может быть весьма полезным. Прежде всего — для часто употребляемых макрокоманд, могут быть включены в библиотеки макроопределений — системные или пользовательские. Это может весьма упростить создание новых макроопределений.

Для обеспечения такой возможности достаточно сделать рекурсивным только 2-й проход Макропроцессора. В нем несколько усложняется анализ операторов макроопределения. В ветви «Другой» (на нашей схеме алгоритма она начинается с блока) 2-й проход Макропроцессора должен распознавать макрокоманду и, если оператор — макрокоманда, вызывать сам себя. Распознавание макрокоманды — методом исключения: если оператор — не оператор Макроязыка, не директива Ассемблера и не машинная команда, то он считается макрокомандой и ищется в Таблице имен макроопределений. Для рекурсивного вызова создается

новая Таблица локальных переменны (и параметров). Таблица глобальных переменных и индекс уникальных меток используются общие.

Некоторая сложность возникает в том случае если вложенные макрокоманды — библиотечные. В нашем алгоритме 1-го прохода содержимое макроопределения (то, что лежит между операторами MACRO и MEND) не анализировалось, следовательно, определения вложенных макрокоманд не заносились в Таблицы макроопределений и имен макропредолений. Есть два варианта решения этой проблемы:

- На 1-м проходе все же распознавать вложенные макровызовы и включать макроопределения их в таблицы.
- Выполнять это на 2-м проходе: при появлении оператора, не распознанного ни как оператор Макроязыка, ни как директива Ассемблера, ни как машинная команда и ни как макрокоманда, определение которой уже есть в наших таблицах, считать его библиотечной макрокомандой и искать ее макроопределение в библиотеках. Если макроопределение найдено, оно добавляется в наши таблицы. Нет необходимости удалять из таблиц определение вложенной библиотечной макрокоманды при завершении обработки внешнего макровызова: оно может потребоваться при обработке и последующих макровызовов.



Качественное расширение возможностей

Активное и грамотное применение макросредств может сделать работу программиста весьма продуктивной. Так, затратив определенные усилия на создание библиотеки макроопределений, программист может превратить язык Ассемблера в качественно новый язык, который будет обладать некоторыми свойствами языка высокого уровня. Программист может сделать этот язык в известной степени проблемно-ориентированным, то есть в максимальной степени приспособленным для тех задач, которые решает его разработчик. Вкратце опишем те основные направления, по которым может идти расширение возможностей Ассемблера за счет макросредств.



Структурный Ассемблер

В виде макрокоманд могут быть реализованы операторы, близкие к операторам управления потоком вычисления в языках высокого уровня (условные операторы, ветвления, различные виды циклов). Известным примером такого расширения является язык Макроассемблера BCPL — предшественник языка C.



Объектно-ориентированный Ассемблер

Макросредства могут обеспечить и реализацию свойств объектно-ориентированного программирования — в большей или меньшей степени.

Простейшее расширение Ассемблера ОО свойствами предполагает введение макрокоманды определения объекта (или резервирования памяти для объекта). В макрокоманде указывается тип объекта и она употребляется вместо директив DC/BSS. Для типа могут быть созданы макрокоманды-операции. В этом варианте может быть воплощен принцип полиморфизма, так как одна и та же операция может быть допустимой для разных типов. (Например, одна команда сложения для всех типов — чисел, независимо от из разрядности и формы представления). Принцип инкапсуляции реализуется здесь в том смысле, что программист, использующий макрокоманды не должен знать внутренней структуры объекта и подробности выполнения операций над ним, защиту же внутренней структуры организовать гораздо сложнее.

Имеются примеры разработок, в которых на уровне Макроязыка созданы и средства описания классов, включающие в себя наследование классов со всеми вытекающими из него возможностями.



Переносимый машинный язык

Макросредствами может быть обеспечен полнофункциональный набор команд некоторой виртуальной машины. Программа пишется на языке этой виртуальной машины. Для разных платформ создаются библиотеки макроопределений, обеспечивающие расширение макровызовов в команды данной целевой платформы. Программа, таким образом, становится переносимой на уровне исходного текста. Поскольку макроопределение может быть построено так, чтобы генерировать неизбыточный код для каждого конкретного вызова, программа на языке виртуальной машины не будет уступать в эффективности программе, сразу написанной на языке целевого Ассемблера.

Лекция 16. Загрузчики и редакторы связей



Основные понятия

Загрузчик — программа, которая подготавливает объектную программу к выполнению и инициирует ее выполнение.

Более детально функции Загрузчика следующие:

- выделение места для программ в памяти (распределение);
- фактическое размещение команд и данных в памяти (загрузка);
- разрешение символических ссылок между объектами (связывание);
- настройка всех величин в модуле, зависящих от физических адресов в соответствии с выделенной памятью (перемещение);
- передача управления на входную точку программы (инициализация).

Не обязательно функции Загрузчика должны выполняться именно в той последовательности, в какой они описаны. Опишем эти функции более подробно.

Функция распределения, по-видимому понятна из ее названия. Для размещения программы в оперативной памяти должно быть найдено и выделено свободное место в памяти.

Для выполнения этой функции Загрузчик обычно обращается к операционной системы, которая выполняет его запрос на выделение памяти в рамках общего механизма управления памятью.

Функция загрузки сводится к считыванию образа программы с диска (или другого внешнего носителя) в оперативную память.

Функция связывания состоит в компоновке программы из многих объектных модулей. Поскольку каждый из объектных модулей в составе программы был получен в результате отдельного процесса трансляции, который работает только с одним конкретным модулем, обращения к процедурам и данным, расположенным в других модулях, в объектных модулях не содержат актуальных адресов. Загрузчик же «видит» все объектные модули, входящие в состав программы, и он может вставить в обращения к внешним точкам правильные адреса. Загрузчики, которые выполняют функцию связывания вместе с другими функциями, называются Связывающими Загрузчиками. Выполнение функции связывания может быть переложено на отдельную программу, называемую Редактором связей или Компоновщиком. Редактор связей выполняет только функцию связывания — сборки программы из многих объектных модулей и формирование адресов в обращениях к внешним точкам. На выходе Редактора связей мы получаем загрузочный модуль.

Функция перемещения необходимо потому, что программа на любом языке разрабатывается в некотором виртуальном адресном пространстве, в котором адресация ведется относительно начала программной секции. При написании программы и при ее трансляции, как правило, неизвестно, по какому адресу памяти будет размещена программа (где система найдет свободный участок памяти для ее размещения). Поэтому в большинстве случаев в командах используется именно адреса меток и данных. Однако, в некоторых случаях в программе возникает необходимость использовать реальные адреса, которые определяться только после загрузки. Все величины в программе, которые должны быть привязаны к реальным адресам, должны быть настроены с учетом адреса, по которому программа загружена.

Существуют программы, которые при написании рассчитываются на размещение в определенных адресах памяти, так называемые, абсолютные программы. Подготовка таких программ к выполнению значительно проще и выполняется она Абсолютным Загрузчиком. Функции такого Загрузчика гораздо проще:

- функция распределения не выполняется, так как реальное адресное пространство, в котором размещается программа предполагается свободным;

- функция загрузки, конечно, выполняется, но она предельно проста;
- функция связывания может быть исключена из Абсолютного Загрузчика: поскольку все адреса программы известны заранее, адреса, по которым происходят обращения к внешним точкам, могут быть определены заранее;
- функция перемещения исключается;
- функция инициализации остается.

Доля абсолютных программ в общей массе программного обеспечения ничтожно мала. Абсолютными могут быть системные программы самого низкого уровня, программы, записываемые в ПЗУ, программы для встраиваемых устройств. Подавляющее же большинство системных и все прикладные программы являются перемещаемыми, то есть, они могут загружаться для выполнения в любую область памяти, и Загрузчик для таких программ выполняет перечисленные функции в полном объеме.

При рассмотрении Ассемблеров мы оставили без внимания обработку обращений к внешним точкам и формат объектного модуля. Эти вопросы непосредственно относятся к функциям Загрузчика, и мы их рассмотрим здесь.

Основные типы Загрузчиков — настраивающие и непосредственно связывающие.

Настраивающие Загрузчики

Настраивающий Загрузчик является первым шагом в сторону усложнения от Абсолютного Загрузчика. Функции связывания и перемещения решаются в нем не самым эффективным, но простейшим способом.

Связывание в Настраивающем Загрузчике

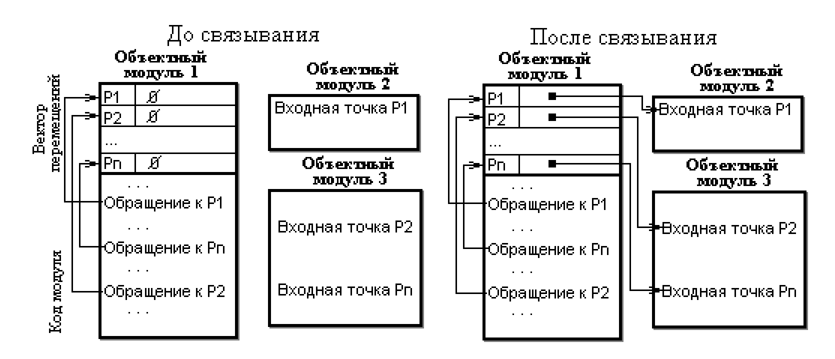
Проблема связывания в Настраивающем Загрузчике решается при помощи Вектора Переходов. Вектор Переходов включается в состав объектного модуля и содержит список всех внешних имен, к которым есть обращение в модуле с полем адреса для каждого имени. Вектор Переходов заполняется при обработке директив типа EXT (перечисления внешних имен).

В команды программы, обращающиеся к внешним именам вставляется обращение к адресному полю соответствующего элемента Векто-

ра Переходов с признаком косвенного обращение. (Косвенное обращение означает, что обращение идет не по адресу, который задан в команде, а по адресу, который записан в ячейке, адрес которой задан в команде.)

При загрузке в оперативную память Вектор Переходов загружается вместе с кодами программы и остается в памяти все время выполнения программы.

Когда Загрузчик компонует программу из нескольких объектных модулей, он «узнает» все фактические адреса входных точек и в Вектора Переходов тех модулей, которые обращаются к данной входной точке вставляет эти адреса. Обращение к внешней точке, таким образом, производится косвенное через Вектор Переходов.



Перемещение в Настраивающем Загрузчике

Принятые в Настраивающих Загрузчиках методы позволяют легко реализовать настройку реальных адресов, заданных относительно начала программы. Сущность метода перемещения состоит в том, что с каждым словом кода программы (размер слова обычно равен размеру реального адреса) связывается «бит перемещения». Значение этого бита 0/1 является признаком неперемещаемого/перемещаемого слова. Если слово является неперемещаемым, оно оставляется Загрузчиком без изменений. Если слово является перемещаемым, то к значению в слове прибавляется стартовый адрес модуля в оперативной памяти. Биты перемещения могут упаковываться — например, описание 8 слов в одном байте.

Непосредственно Связывающие Загрузчики

Эти Загрузчики называются непосредственно связывающими потому, что они обеспечивают обращение к внешней точке непосредственно, а не через косвенную адресацию. Эти Загрузчики обеспечивают более

высокую эффективность кода и более гибкие возможности связывания. Такие возможности достигаются за счет того, что в объектном модуле содержится вся необходимая для Загрузчика информация.



Формат объектного модуля

Объектный модуль, поступающий на вход Загрузчика должен в той или иной форме содержать:

- размер модуля;
- машинные коды;
- входные точки (те адреса в модуле, к которым возможны обращения извне);
- внешние точки (те имена во внешних модулях, к которым есть обращения в данном модуле);
- информация о размещении в модуле перемещаемых данных.

Объектный модуль состоит из записей четырех типов. В каждой записи первый байт содержит идентификатор типа записи, следующие 2 байта — размер записи. Количество и формат следующих байтов записи определяется ее типом.

Кодовая запись содержит адрес относительно начала модулей и коды программы. Кодовые записи строятся Ассемблером при генерации объектного кода — кодов команд и директив типа DC. Каждая кодовая запись начинается с адреса (относительно начала модуля), с которого начинается размещение ее содержимого. Разрывы в линейном пространстве адресов могут быть обусловлены директивами выделения памяти без записи в нее значений (директивы типа BSS) или разрывами, управляемыми программистом (директивы типа ORG).

Запись связываний содержит один или несколько элементов Таблицы внешних символов. Элемент Таблицы внешних символов имеет следующий формат:

Имя символа	Тип символа: • имя сегмента • входная точка • внешняя точка.	Относительный адрес в модуле (только для сегментов и внешних точек)	Длина (только для сегментов)

Ассемблер формирует новые элементы Таблицы перемещений при обработке директив типа SEGMENT, ENT, EXT.

Запись перемещений содержит один или несколько элементов Таблицы перемещений. Каждый элемент этой таблицы описывает один элемент кода программы, требующий настройки, зависящей от адреса загрузки программы в память и имеет следующий формат:

Относительный адрес в модуле	Имя символа	Бит операции	Длина

Относительный адрес и длина — адрес и длина того кода, который должен быть модифицирован. Имя символа — имя из Таблицы внешних символов, которое прибавляется к значению кода или вычитается из него. Бит операции — признак операции сложения или вычитания. Ассемблер генерирует элемент Таблицы перемещений, когда обрабатывает адресные выражения. Адресное выражение может быть абсолютным (независящими от адреса загрузки) или перемещаемым (зависящими от адреса загрузки). Элементы Таблицы перемещений генерируются только для перемещаемых выражений. Рассмотрим адресное выражение:

```
ADDR1 – ADDR2
```

Если ADDR1 и ADDR2 являются именами, перемещаемыми внутри одного и того же сегмента SEGM (их адресные значения определяются относительно начала сегмента), то адресное выражение является абсолютным, так как его значение: SEGM+ADDR1 — (SEGM+ ADDR2) не зависит от адреса сегмента. Для такого выражения элемент Таблицы перемещений не строится.

Если ADDR1 является именем, перемещаемым внутри сегмента SEGM, а ADDR2 — абсолютный адрес, то выражение является простым перемещаемым. В кодовое представление этого выражения записывается разность между относительным адресом в сегменте ADDR1 и абсолютным значением ADDR2. Для такого выражения строится элемент Таблицы перемещений:

```
адрес   SEGN   +   длина
```

Загрузчик прибавит к содержимому кода адрес сегмента SEGM.

Если ADDR1 является внешним именем, а ADDR2 — абсолютный адрес, то выражение также является простым перемещаемым. В кодовое представление этого выражения абсолютное значение ADDR2. Элемент Таблицы перемещений для такого выражения содержит:

```
адрес   ADDR1   +   длина
```

Если ADDR1 и ADDR2 являются внешним именем, то выражение является сложным перемещаемым. В кодовое представление этого выражения записывается 0. Для такого выражения строятся два элемента Таблицы перемещений:

```
адрес   ADDR1   +   длина
адрес   ADDR2   –   длина
```

При загрузке к нулевому значению записанному по адресу адрес будет прибавлен адрес внешней точки ADDR1, а затем вычтен адрес внешней точки ADDR2.

Запись окончания формируется Ассемблером при обработке директивы END, она содержит стартовый адрес программы. Естественно, эта запись должна быть заполнена только в одном из объектных модулей, составляющих программу.



Алгоритм работы Непосредственно Связывающего Загрузчика

Наиболее простой алгоритм работы Загрузчика — двухпроходный.

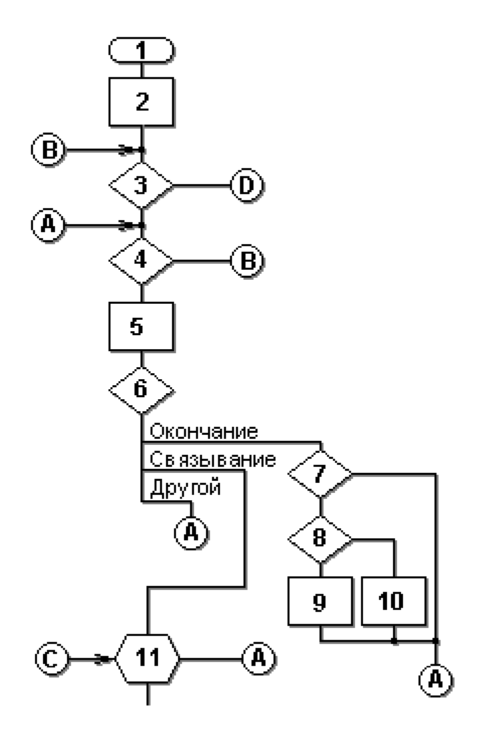
На вход Загрузчика обязательно подается список объектных модулей, из которых составляется программа. Этот список может быть параметром программы-Загрузчика или находиться в отдельном файле. На первом проходе Загрузчик просматривает все объектные модули по списку и решает 2 задачи:

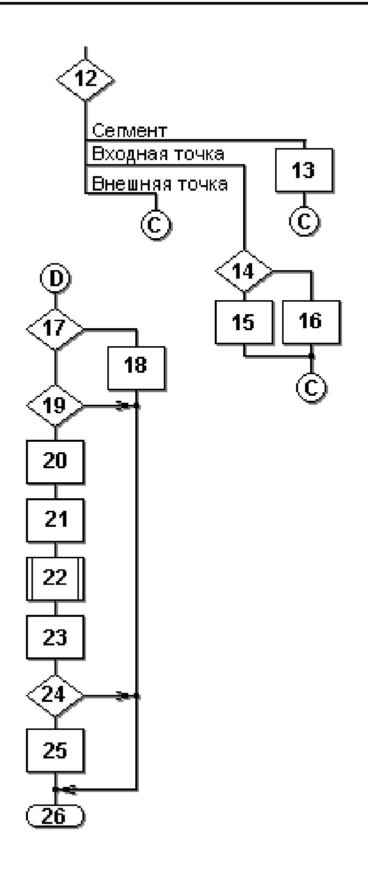
- определяет общий объем области памяти, необходимый для программы и размещение объектных модулей в этой области;
- составляет Глобальную таблицу внешних имен программы.

Структура элемента Глобальной таблицы — такая же, как и Таблицы внешних символов каждого модуля. В нее заносятся только входные точки всех модулей. Поскольку Загрузчик уже знает в каком месте области памяти, выделяемой для программы, будет размещаться тот или иной модуль, он заносит в Глобальную таблицу адреса входных точек относительно начала всей программы. В конце 1-го прохода Загрузчик выделяет память и, уже зная фактический начальный адрес программы в памяти корректирует все адреса в Глобальной таблице внешних символов.

На 2-м проходе Загрузчик снова читает все объектные модули по списку. При этом он уже размещает коды модуля в памяти и формирует для каждого модуля Таблицу внешних символов (локальную для модуля) и Таблицу перемещений. Адресные поля в этих таблицах он заполняет или корректирует с учетом фактического адреса модуля в памяти и содержимого Глобальной таблицы внешних символов. Затем он выполняет обработку Таблицы перемещений, используя для коррекции адресных кодов в программе значения из Локальной Таблицы внешних символов.

Алгоритм выполнения 1-го прохода — следующий:



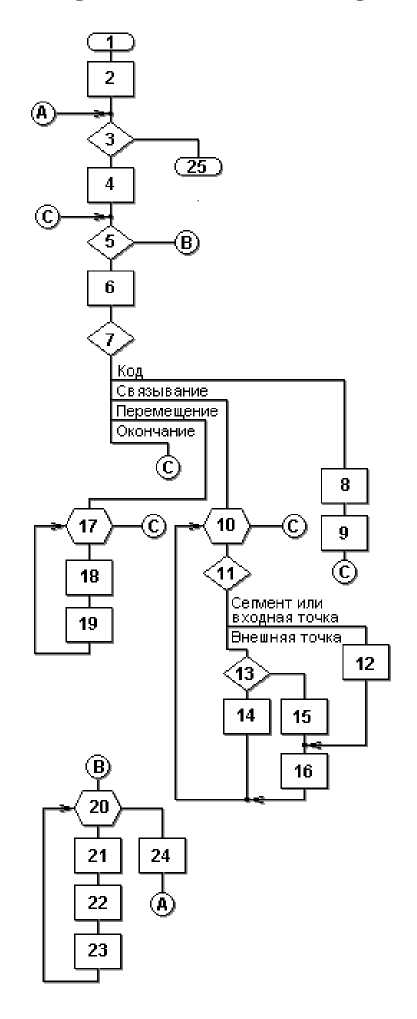


- **Блок1:** 1-й проход Загрузчика.
- **Блок2:** Начальные установки. Создание пустой Глобальной таблицы. Стартовый адрес=пусто. Относительный адрес 1-го сегмента — 0. Размер программы — 0.
- **Блок3:** Выборка следующего имени из списка объектных модулей. Если весь список объектных модулей обработан — переход на окончание 1-го прохода.
- **Блок4:** Чтение заголовка очередной записи объектного модуля, если объектный модуль обработан полностью — переход к следующему модулю.
- **Блок5:** Чтение остальной части записи (размер записи содержится в ее заголовке).

- **Блок6:** Разветвление в зависимости от типа записи.
- **Блок7:** При обработке записи окончания проверяется, имеется ли в записи стартовый адрес. Если стартового адреса нет — никакая другая обработка записи не производится.
- **Блок8:** Если в записи есть стартовый адрес, проверяется, не был ли он уже установлен.
- **Блок9:** Если стартовый адрес не был установлен, он устанавливается.
- **Блок10:** Если стартовый адрес был установлен, выдается сообщение об ошибке. (Ни эта, ни последующие рассмотренные ошибки не приводят к немедленному завершению 1-го прохода, однако, если на 1-м проходе были ошибки, 2-й проход не выполняется).
- **Блок11:** При обработке записи связывания выполняется перебор элементов Таблицы внешних символов...
- **Блок12:** ...и разветвление — в зависимости от типа элемента.
- **Блок13:** Для элемента — сегмента вычисляется начальный адрес следующего сегмента и длина сегмента прибавляется к общему размеру программы.
- **Блок14:** Для элемента — входной точки ищется имя точки в Глобальной таблице.
- **Блок15:** Если имя не найдено в Глобальной таблице, в таблицу добавляется новый элемент.
- **Блок16:** Если имя найдено в Глобальной таблице, — ошибка, неуникальное внешнее имя.
- **Блок17:** При окончании 1-го прохода проверяется, установился ли адрес стартовой точки программы.
- **Блок18:** Если этот адрес не установлен — ошибка.
- **Блок19:** Если этот адрес установлен и в ходе выполнения 1-го прохода не было других ошибок, Загрузчик продолжает работу.
- **Блок20:** Выделяется память для программы в соответствии с ее размером.
- **Блок21:** В Глобальную таблицу внешних символов записываются фактические адреса.

- **Блок22:** Выполняется 2-й проход.
- **Блок23:** Освобождается Глобальная таблица
- **Блок24:** Если не было ошибок на 2-м проходе
- **Блок25:** ...управление передается на стартовый адрес программы
- **Блок26:** Загрузчик завершает работу.

Алгоритм выполнения 2-го прохода — следующий:



- **Блок1:** 2-й проход Загрузчика
- **Блок2:** Установка на начало списка имен объектных модулей.
- **Блок3:** Выборка следующего имени из списка объектных модулей. Если весь список объектных модулей обработан — переход на окончание 2-го прохода.
- **Блок4:** Создание для модуля Локальной таблицы внешних символов и Таблицы перемещений — пустых.
- **Блок5:** Чтение заголовка очередной записи объектного модуля, если все записи модуля прочитаны — переход к обработке перемещений в модуле.
- **Блок6:** Чтение остальной части записи (размер записи содержится в ее заголовке).
- **Блок7:** Разветвление в зависимости от типа записи.
- **Блок8:** Для кодовой записи считывается относительный адрес записи и переводится в фактический.
- **Блок9:** Тело кодовой записи считывается и размещается в памяти по фактическому адресу.
- **Блок10:** Для записи связывания перебираются находящиеся в ней элементы Таблицы имен
- **Блок11:** Обработка разветвляется в зависимости от типа имени.
- **Блок12:** Для имен сегментов или входных точек относительный адрес переводится в фактический.
- **Блок13:** Имя внешней точки ищется в Глобальной таблице внешних имен.
- **Блок14:** Если имя не найдено в Глобальной таблице, выдается сообщение об ошибке.
- **Блок15:** Если имя найдено в Глобальной таблице, в значение адреса из Глобальной таблицы становится значением этого имени.
- **Блок16:** Элемент с откорректированным адресом заносится в Локальную таблицу имен.
- **Блок17:** Для записи перемещения перебираются находящиеся в ней элементы Таблицы перемещения.
- **Блок18:** Относительный адрес в элементе заменяется на фактический...

- **Блок19:** ...и элемент добавляется в Таблицу перемещений.
- **Блок20:** После того, как весь модуль прочитан, выполняется перебор Таблицы перемещений модуля.
- **Блок21:** Для каждого элемента Таблицы перемещений имя, записанное в его поле имени ищется в Локальной таблице имен и из Локальной таблицы имен выбирается связанный с этим именем адрес.
- **Блок22:** Из кода программы выбирается код, адрес и длина которого записаны в элементе Таблицы перемещений.
- **Блок23:** Над выбранным кодом и адресом, выбранным из Таблицы имен выполняется операция сложения или вычитания, результат записывается на место кода.
- **Блок24:** После перебора всей Таблицы перемещений, освобождаются Таблица перемещений и Локальная таблица имен модуля, и управление передается на обработку следующего модуля.
- **Блок25:** После обработки всех объектных модулей 2-й проход заканчивается.

Лекция 17. Кросс-системы



Вычислительные системы

Исходная вычислительная система (ВС) — та ВС, на которой программа готовится к выполнению.

Целевая ВС — та ВС, на которой программа выполняется.

Эти две ВС не обязательно совпадают. Макропроцессор, Ассемблер, Редактор Связей — программы, обрабатывающие данные. Ассемблер, например, получает на входе одни код (текст) и производит на выходе другой код (объектный модуль). При этом Ассемблер не рассматривает свой выходной код как команды именно своей ВС, это просто некоторые данные. Ничто не мешает нам сделать Ассемблер, на выходе которого будут генерироваться коды не той ВС, в которой работает Ассемблер, а некоторой другой ВС.

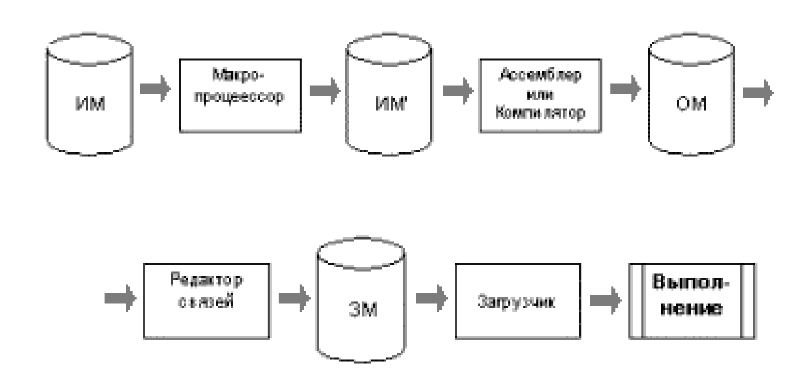
Системы подготовки программ, в которых исходная ВС отличается от целевой, называются кросс-системами.

Для чего может понадобиться кросс-система?

Часто кросс-системы применяются для разработки программного обеспечения встроенных вычислительных систем. Для встроенных ВС характерен малый объем ресурсов: ограничение оперативной памяти, возможно, отсутствие внешней памяти (программы и постоянные данные размещаются в ПЗУ), отсутствие должного набора внешних устройств. Иногда ресурсов целевой ВС просто недостаточно для выполнения на ней системного программного обеспечения подготовки программ, тем более — для выполнения интерактивных систем программирования с развитым интерфейсом пользователя.

При разработке новых ВС создание программного обеспечения для них ведется параллельно с разработкой аппаратной части. Подготовка и отладка программ для проектируемой ВС должна вестись, когда целевой ВС еще не существует физически.

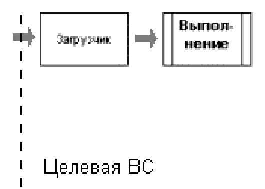
В простейшем случае процесс подготовки программы на кросс-системе аналогичен процессу их подготовки в однородной системе.



Загрузка и выполнение должна обязательно происходить на целевой системе, а предшествующие этапы (все или некоторые) могут быть перенесены на исходную систему. Любой из файлов-результатов выполнения очередного этапа подготовки может быть перенесен из исходной системы в целевую.

Однако, такая схема не в полной мере использует возможности кросс-системы. Важным этапом подготовки программ является их отладка. Отладка также может проводиться на кросс-системе (частично или полностью)





Для отладки программ на исходной ВС применяется программа-Интерпретатор. Интерпретатор является программной моделью целевой ВС, которая обеспечивает выполнение программ в кодах целевой ВС. Интерпретатор строится по принципу имитационной программной модели, это означает, что отдельные компоненты целевой ВС моделируются соответствующими компонентами программной модели, которые имитируют поведение реальных компонентов.

Если исходная ВС обладает большими вычислительными ресурсами, чем целевая ВС, то отладка на исходной ВС может быть более удобной и функционально более полной, чем на целевой ВС. Это, впрочем, относится и к тому случаю, когда исходная ВС не превосходит целевую по объему ресурсов. В таком случае для отладки программы все равно может быть выделено больше ресурсов (возможно, виртуальных), чем при ее выполнении.

Модель, на которой производится отладка, всегда является избыточной по ресурсам по сравнению с целевой средой выполнения. Избыточность модели позволяет выявить при отладке на ней такие ошибочные ситуации, которые трудно или вообще невозможно выявить при отладке в реальной среде. Примеры таких ситуаций:

- обращение по адресу несуществующей памяти
- попытка записи в защищенную от записи память
- модификация программой команд и констант
- передача управления на данные
- выборка неинициализированных данных

Модель целевой вычислительной системы состоит из компонентов, моделирующих программно-доступные компоненты целевой ВС (т.е. такие, с которыми работают команды отлаживаемой программы) и включает в себя следующие составляющие:

- модель регистров
- модель оперативной памяти

- модель процессора
- модель системы прерывания
- модель системы ввода-вывода.



Модель регистров

Модель регистров включает в себя, как минимум:

- регистры общего назначения
- регистр-счетчик адреса
- регистр состояния

Регистры моделируются переменными интерпретатора.

РОН во время выполнения программы содержат обрабатываемые данные. РОН могут моделироваться как отдельными переменными, так и массивами — в зависимости от их количества и свойств.

В тех ВС, где РОН немного и некоторые из них обладают собственными индивидуальными свойствами (напр., Intel) удобно представлять каждый РОН в виде отдельной переменной.

Для тех ВС, где РОН много и/или они одинаковы во всем (напр., S/390, все RISC), их целесообразно представлять в виде массива. Характерно, что в ВС первого типа РОН обычно имеют собственные имена, а в ВС второго типа РОН идентифицируются номерами.

Счетчик адреса содержит адрес текущий выполняемой команды и представляется в виде отдельной переменной.

Регистр состояния содержит признаки результата выполнения предыдущей команды — больше, меньше, равен нулю (не все команды устанавливают эти признаки) и, возможно, признак привилегированного/непривилегированного режима. Эти признаки могут «упаковываться» в одну переменную или представляться отдельной переменной каждый.



Модель оперативной памяти

Объем адресного пространства памяти, к которому теоретически могут выполняться обращения к программе определяется разрядностью представления адреса. Однако, реально в целевой ВС может быть значительно меньший объем памяти. Во встроенных ВС адресное пространство может покрываться реальной памятью несмежными фрагментами, причем фрагменты реальной памяти могут быть как ОЗУ, так и ПЗУ.

Интерпретатор должен «знать» конфигурацию реальной памяти в целевой ВС. Возможные варианты задания такой конфигурации:

Потребовать, чтобы любая ячейка памяти, к которой обращается программа, была описана в программе (директивой DD или BSS).

Описать конфигурацию памяти в отдельном файле, являющимся входным для Интерпретатора.

Представляется, что второй подход более универсальный, так как:

- обращение в программе по неописанному в ней адресу памяти возможно (особенно это касается программ для встроенных ВС с абсолютными программами и жестки распределением памяти);
- определение памяти в программе также является объектом проверки/отладки может содержать ошибки;
- в Ассемблере нет средств описания ОЗУ/ПЗУ.

Внешнее описание памяти считывается Интерпретатором в начале работы и превращается в таблицу фрагментов.

Оперативная память целевой ВС представляется памятью (не обязательно оперативной) исходной ВС. Однако, в модели памяти на исходной ВС мы имеем возможность помимо собственно данных, хранящихся в целевой памяти, представлять также и описание этих данных.

Каждый байт целевой памяти представляется двумя байтами исходной памяти. В первом байте представления хранятся собственно данные, а во втором — ряд признаков, характеризующих ячейку целевой памяти.

Среди этих признаков могут быть такие:

- a) признак 1-го байта команды (управление можно передавать только на 1-й байт команды);
- b) признак команды/данных
- c) признак инициализированных/неинициализированных данных
- d) признак изменяемых/неизменяемых данных
- e) признак останова при передаче управления
- f) признак останова при передаче записи
- g) признак останова при передаче чтении

Все названные признаки — однобитные. Признаки a, b устанавливаются Кросс-ассемблером при трансляции программы и не изменяются при выполнении. Признак c устанавливается Кросс-ассемблером, но может изменяться Интерпретатором в процессе выполнения. Признак d устанавливается Интерпретатором перед началом выполнения на основе таблицы фрагментов и, возможно, дополнительной информации, вводимой программистом (отдельно от программы) и может изменяться программистом в ходе интерактивной отладки. Признаки e-f устанавливаются перед началом выполнения на основе дополнительной информации и может изменяться программистом в ходе интерактивной отладки.

Дополнительная информация о памяти, таким образом, состоит из таблицы фрагментов, списка переменных в ОЗУ, которые не разрешается изменять, списка переменных, при обращении к которым должен происходить останов, и меток, при передаче управления на которые должен происходить останов.

Каждое обращение к памяти в программе характеризуется типом: R (чтение), W (запись) или X (передача управления). При любом типе обращения проверяется попадание в реально существующий фрагмент памяти. При обращении типа X проверяется бит a признака, управление может быть передано только на байт с установленным признаком a. При обращениях типа R и W проверяется бит b признака, обращения этого типа могут происходить только к данным При обращениях типа R проверяется бит c признака, читаться могут только инициализированные данные При обращениях типа W проверяется бит d признака, данные должны быть изменяемые, бит c признака при этом устанавливается, то есть, данные становятся инициализированными.



Модель процессора

Работа процессора моделируется алгоритмом работы Интерпретатора. Основной алгоритм работы модели состоит из цикла, в каждой итерации которого моделируется выполнение одной команды целевой программы. Итерация этого цикла начинается с выборки байта, записанному в модели памяти по адресу, содержащемуся в модели регистра-счетчика адреса. В подавляющем большинстве ВС первый байт команды содержит код операции, позволяющий однозначно идентифицировать команду. Интерпретатор выполняет поиск по коду операции в таблице команд.

При этом может использоваться либо таблица команд Ассемблера, либо ее модификация с расширениями и с возможностью быстрого поиска по коду операции. Распознав команду, Интерпретатор выбирает ее остальные байты (их количество определено в таблице команд) и выделяет из них операнды команды (их количество и кодировка определяется типом команды).

Далее алгоритм Интерпретатора разветвляется, в общем случае число ветвей равно числу возможных кодов операции.

В каждой ветви вычисляется значение, являющееся результатом выполнения той или иной команды. Вычисленное значение заносится в объект, являющийся для данной команды приемником результата.

Кроме того для тех команд, для которых это требуется устанавливаются значения признаков в регистре состояния (перечень признаков, устанавливаемых командой, может содержаться в таблице команд Интерпретатора). Вычисляется новое значение регистра-счетчика адреса. В большинстве случаев это значение получается добавлением к текущему его значению длины команды, но в командах перехода (типа JMP, CALL) это значение вычисляется.

При реализации алгоритмов выполнения отдельных команд возможны два подхода, которые мы называем RISC и CISC-моделями, по аналогии с архитектурами процессоров (однако выбор программной RISC или CISC-модели необязательно должен совпадать с реальной архитектурой процессора).

Смысл RISC-модели состоит в том, что разветвление алгоритма выполняется сразу же после распознавания команды и выполнение каждой команды полностью реализуется кодами соответствующей ветви.

Смысл CISC-модели состоит в том, что выполнение каждой команды представляется в виде последовательности выполнения простых процедур («микрокоманд»). Список микрокоманд, составляющих выполнение каждой команды, может быть «зашит» в программу в виде последовательности вызовов или представлен в виде данных, например, в виде списка номеров процедур. В последнем случае алгоритм не требует ветвления, а сводится к циклу, в каждой итерации которого выбирается номер очередной процедуры и вызывается процедура с данным номером. В предельном случае выполнение каждой команды может быть представлено в виде исходного текста на языке макрокоманд, который интерпретируется Интерпретатором.

Пример

Пусть в языке микрокоманд имеются следующие (показаны не все) микрокоманды:

GETR n,rx

Выборка номера регистра, заданного в n-ом операнде в промежуточную переменную rx

GETA n,ax

Выборка адреса, заданного в n-ом операнде в промежуточную переменную ax

LDR dx,rx

Выборка данных из регистра, номер которого находится в промежуточной переменной rx в промежуточную переменную dx

LDM dx,ax

Выборка данных из памяти по адресу, находящемуся в промежуточной переменной ax промежуточную переменную dx

SDR rx,dx

Запись данных из промежуточной переменной dx в регистр, номер которого находится в промежуточной переменной rx

SDM dx,ax

Запись данных из промежуточной переменной dx в память по адресу, находящемуся в промежуточной переменной ax промежуточную переменную dx

ADD dx1,dx2

Сложение данных из промежуточной переменной dx1 с данными из dx2; результат — в dx1

SIG dx

Инверсия знака данных, содержащихся в промежуточной переменной dx

CC1 dx

Установка признаков «больше», «меньше», «равно» по значению, содержащемуся в промежуточной переменной dx

CC2 dx

Установка признака переполнения по значению, содержащемуся в промежуточной переменной dx

PC1

Увеличение регистра-счетчика адреса на длину команды

PC2 dx

Запись данных из промежуточной переменной dx в регистр-счетчика адреса

END

Окончание микропрограммы

Тогда реализация некоторых машинных команд может быть «замикропрограммирована» следующим образом:

LR регистр2,регистр1

Пересылка данных из регистра1 в регистр2

```
GETR 2, r1;
LDR d1,r1;
GETR 1,r2;
SDR r2,d1,
PC1;
END;
```

L регистр,память

Пересылка данных из памяти по адресу память в регистр

```
GETA 2,a1;
LDM d1,a1;
GETR 1,r1;
SDR r1,d1;
```



```
PC1;
END;
```

AR регистр2,регистр1

Сложение данных из регистра1 с данными в регистре2; результат — в регистре1

```
GETR 2, r1;
LDR d1,r1;
GETR 1,r2;
LDR d2, r2;
ADD d1,d2;
SDR r1;
PC1;
END;
```

CMP регистр,память

Сравнение данных, содержащихся в регистре с данными по адресу память

```
GETR 1,r1;
LDR d1,r1;
GETA 2, a1;
LDM d2,a1;
SIG d2;
ADD d1,d2;
CC1 d1;
CC2 d1;
PC1;
END;
```

JMP память

Переход по адресу память

```
GETA 2,a2;
PC2 a2;
END;
```

Очевидно, что RISC-модель будет выполняться быстрее, но CISC-модель гибче, так как активные элементы (команды) в ней превращены в пассивные (данные). В аппаратных архитектурах предпочтение отдается RISC из-за высшей эффективности, а какие критерии являются более важными при отладке?

Время

Для значительного класса встроенных ВС время выполнение программы является принципиально важной ее характеристикой (напри-

мер, бортовые системы управления должны работать в реальном времени).

Важно понимать, что время выполнения программы на Интерпретаторе ни в коей мере не соответствует времени ее выполнения на реальной ВС. Более того, временные соотношения между выполнением различных частей программы на модели также не соответствуют соотношениям выполнения частой программы на реальном оборудовании. Поэтому время также является моделируемым компонентом. Моделью времени является целая переменная большой разрядности. В этой переменной на каждом шаге выполнения содержится число машинных тактов, выполненных с начала выполнения программы. Исходное значение этой переменной — 0, после выполнения каждой команды ее значение увеличивается на время выполнения данной команды (время выполнения может быть столбцом в таблице команд).

Система прерываний

Является самым сложным для моделирования компонентом. Трудность состоит в том, что прерывания поступают асинхронно, без привязки к выполнению программы. Следовательно, прерывания должны «зарождаться» где-то вне собственно выполняемой программы.

При выполнении Интерпретатора в пошаговом режиме прерывания могут задаваться командами, вводимыми человеком-оператором. Более универсальным является прием, предполагающий создание в отдельном файле «программы поступления прерываний». Каждый «оператор» этой «программы» содержит идентификатор типа прерывания и время (модельное) поступления прерывания. Эти «операторы» должны быть упорядочены по возрастанию времен поступления. Поскольку ВС обладают свойством непрерываемости команд, условие поступления прерывания может проверяться только после окончания обработки очередной команды.

Действия по прерыванию определяются характеристиками конкретной ВС. Как правило, они включают в себя запоминание текущего значения регистров состояния и счетчика адреса и занесение в счетчик адреса адреса программной секции обработки прерывания данного типа.

Отладке программ, предусматривающих обработку внешних прерываний, усложняется многократно, так как при этом должно быть предусмотрено поступление внешних прерываний во все возможные (и невозможные!) моменты выполнения.

Ввод-вывод

Операции ввода-вывода целевой ВС моделируются файловым вводом-выводом исходной ВС. Данные, которые целевая ВС вводит с внешнее устройство, читаются моделью из файла. Данные, которые целевая ВС выводит на внешнее устройство, записываются моделью в файл. Для каждого внешнего устройства удобно назначать свой файл. В частном случае это может быть файл клавиатуры или файл экрана. Данные в файл, имитирующий устройство ввода, должны быть занесены заранее. На вход Интерпретатора должна подаваться таблица соответствия файлов устройствам.

Ввод-вывод может быть синхронным или асинхронным. При синхронном вводе-выводе (например, через порты) операция ввода-вывода завершается вместе с завершением команды ввода-вывода. Моделирование такого ввода-вывода сложностей не представляет. При асинхронном вводе-выводе (КПДП, каналы ввода-вывода) команда ввода-вывода только запускает операцию ввода-вывода и заканчивается. Выполнение операции ввода-вывода далее происходит параллельно с выполнением команд программы, а об окончании ввода-вывода устройство сигнализирует прерыванием. И здесь «срабатывают» трудности, присущие моделированию системы прерываний. Одно из возможных решений — при инициализации операции ввода-вывода добавлять в программу поступления прерываний новый элемент, соответствующий прерыванию, которое поступит через какой-то интервал времени после текущего момента.

Особую сложность представляет собой моделирование ошибочных ситуаций ввода-вывода, эта проблема должна решаться для каждого прикладного случая, поэтому здесь не рассматривается.

Взаимодействие с человеком-оператором

Интерпретатор может выполняться в автоматическом или пошаговом режиме. В автоматическом режиме Интерпретатор моделирует выполнение команд программы без остановок до команды типа HALT или до точки останова. В точке останова оператор может вводить команды, управляющие действиями Интерпретатора и выбрать режим продолжения выполнения. В пошаговом режиме Интерпретатор после выполнения каждой команды программы останавливается и предоставляет оператору возможность вводить команды управления. Командами управления работой Интерпретатора могут быть:

- команды на отображение/изменение состояния/содержимого компонентов модели;
- команды задания точек останова;

- команды моделирования прерываний;
- команды установки режима выполнения;
- команда окончания работы.

Отображаться должны состояния и значения всех составляющих программной модели ВС: регистров (РОН, счетчика адреса, состояния), заданных участков памяти и их признаков, счетчика модельного времени, программы поступления прерываний и т.д. Отображаемые значения также должны быть доступны для изменений. Отметим, что для интерактивного отображения/изменения должны быть доступны также байты признаков памяти. Изменение содержимого регистра-счетчика адреса равносильно передачи управления в программе.

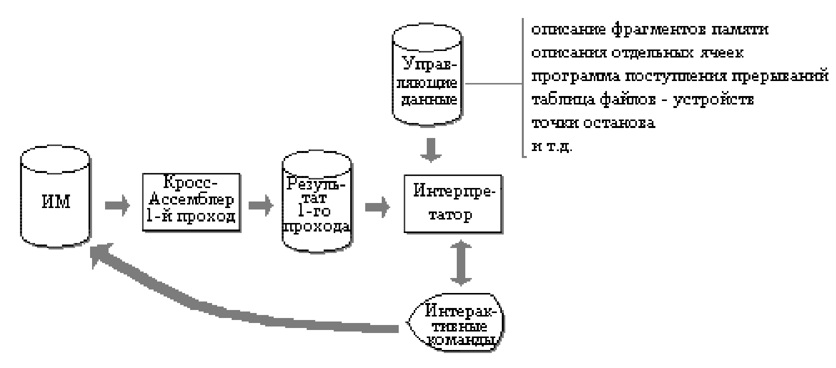
Точки останова могут задаваться в исходном для Интерпретатора файле и вводиться/изменяться в ходе интерактивной отладки. Могут быть предусмотрены остановы при:

- передаче управления по заданному адресу;
- чтении данных по заданному адресу;
- записи данных по заданному адресу.

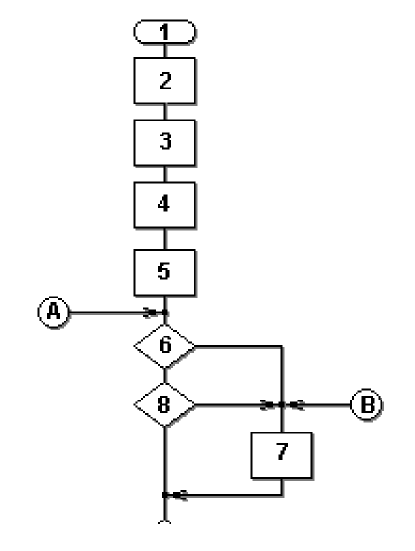
Связь отладки с исходным текстом. Такая связь безусловно удобно и может быть осуществлена относительно несложно, если выход 1-го прохода Кросс-Ассемблера передается на вход Интерпретатора. Выход 1-го прохода связывает операторы исходного текста с адресами памяти. Таким образом, по значению счетчика адреса в каждый момент выполнения программы можно найти в выходе 1-го прохода соответствующий оператор исходного текста. Если на вход Интерпретатора подается также сформированная 1-м проходом таблица символов, то есть возможность обращаться к переменным программы и к точкам передачи управления по символьным именам.

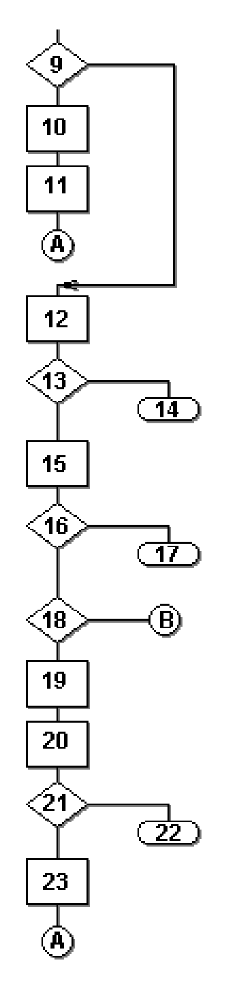
Можно ли обеспечить изменение прямо в ходе отладки исходного текста? Схема решения сводится к представленной на рисунке. В схеме остается только 1-й проход Кросс-Ассемблера. Выход его — исходный текст с разметкой адресов и таблица символов является основным входом Интерпретатора. Необходимость во 2-м проходе Кросс-Ассемблера отпадает. В начале выполнения Интерпретатор должен построить модель памяти, в которой он размещает, однако, только данные программы, но не команды. При работе Интерпретатор повторяет многие действия 2-го прохода Кросс-Ассемблера, читает не коды, а исходные тексты и распознает команду не по коду операции, а по мнемонике, и интерпретирует операнды не по кодам, а по исходным текстам. Изменения в исходном

тексте оператора программы должны автоматически реплицироваться в соответствующем операторе (только в одном операторе!) результата 1-го прохода, и тогда при следующем выполнении этого оператора будет моделироваться уже выполнение новой команды. Однако, поскольку в результате 1-го прохода каждый оператор уже привязывается к определенному адресу, возможность изменения должна ограничиваться тем, что длина новой команды обязательно равна длине старой команды. Более сложные изменения потребуют повторного выполнения 1-го прохода Кросс-Ассемблера.



Итоговая схема алгоритма функционирования Интерпретатора сводится к следующей:





- **Блок1:** Запуск Интерпретатора.
- **Блок2:** Открытие исходных файлов — результатов работы Кросс-Ассемблера и файлов с управляющей информацией (описание файлов — внешних устройств, программа поступления прерываний, описание фрагментов памяти и отдельных ячеек и т.п.).
- **Блок3:** Считывание управляющей информации.
- **Блок4:** Установка начальных значений для компонентов модели (содержимое памяти, регистры, счетчик модельного времени).

- **Блок5:** Интерактивное задание/корректировка управляющей информации (режим выполнения, точки останова и т.п.).
- **Блок6:** Автоматический режим?
- **Блок7:** Если установлен пошаговый (не автоматический) режим выполнения, выполняется ввод и обработка команд оператора в интерактивном режиме. Эта обработка может заканчиваться либо продолжением выполнения интерпретатора в пошаговом или автоматическом режиме, либо завершением его работы по команде оператора.
- **Блок8:** Если установлен автоматический режим выполнения, но текущее значение регистра — счетчика адреса совпадает с одной из заданных точек останова, также выполняется ввод и обработка команд оператора в интерактивном режиме.
- **Блок9:** Проверяется счетчик модельного времени сравнивается с временем поступления первого прерывания в списке прерываний.
- **Блок10:** Если счетчик модельного времени больше или равен времени поступления первого прерывания в списке, выполняется сохранение текущего состояния и занесение в регистр-счетчик адреса секции обработки прерывания данного типа.
- **Блок11:** Первый элемент удаляется их списка прерываний и происходит возврат на начало итерации обработки команды.
- **Блок12:** Если прерывание не поступило, выбирается первый байт команды (при отладке по объектному модулю) или ее мнемоника (при отладке по исходному тексту).
- **Блок13:** Код операции или мнемоника команды ищется в таблице команд.
- **Блок14:** При неуспешном поиске Интерпретатор заканчивается с сообщением об ошибке.
- **Блок15:** Выбор операндов из кода команды или из текста оператора.
- **Блок16:** Проверка правильности кодирования операндов, проверка корректности обращения к памяти.
- **Блок17:** При ошибках в операндах или в обращении к памяти Интерпретатор заканчивается с сообщением об ошибке.

- **Блок18:** Задан ли для адреса операнда останов при обращении? Если да — возврат на выполнение команд в интерактивном режиме.
- **Блок19:** Интерпретация команды и запись результата
- **Блок20:** Вычисление и занесение в регистр-счетчик адреса следующей команды.
- **Блок21:** Проверка, является ли адрес в регистре-счетчике адреса адресом 1-го байта команды
- **Блок22:** Если это не так, Интерпретатор заканчивается с сообщением об ошибке.
- **Блок23:** Модификация счетчика модельного времени и переход на выполнение следующей команды.

Окончание работы Интерпретатора может происходить:

- при обнаружении ошибки в программе;
- при вводе оператором интерактивной команды завершения работы;
- при обработке команды останова (HALT) в программе.

Лекция 18. Ошибки программирования



Классификация ошибок программирования

Рассмотренные ошибки программирования могут быть разделены на следующие категории:

Перестановка операндов или частей операндов

К типичным ошибкам этого рода относятся:

- перестановка операндов, указывающих на источник и назначение в командах пересылки;
- перевертывание формата, в котором запоминаются 16-разрядные значения;
- изменение направления при вычитаниях и сравнениях.

Неправильное использование флагов

Типичные ошибки следующие:

- использование не того флага, который в данном конкретном случае должен проверяться (как, например, флага знака вместо флага переноса);
- условный переход после команд, которые не воздействуют на данный флаг;
- инвертирование условий перехода (особенно при использовании флага нуля);

- неправильный условный переход в случаях равенства и случайное изменение флага перед условным переходом.

Смешивание регистров и пар регистров

Типичная ошибка состоит в работе с регистром (**B**, **D** или **H**) вместо пары регистров с аналогичным именем.

Смешивание адресов и данных

К типичным ошибкам относятся:

- использование непосредственной адресации вместо прямой адресации или наоборот;
- смешивание регистров с ячейками памяти, адресуемыми через пары регистров.

Использование неверных форматов

Типичные ошибки состоят в:

- использовании формата **BCD** (десятичного) вместо двоичного или наоборот,
- использование двоичного и шестнадцатеричного кода вместо **ASCII**.

Неправильная работа с массивами

Обычная ошибка состоит в выходе за границы массивов.

Неучет неявных эффектов

К типичным ошибкам относятся использование без учета влияния участвующих в работе команд:

- аккумулятора;
- пары регистров;
- указателя стека;
- флагов или ячеек памяти.

Большинство ошибок вызываются командами, которые дают непредвиденные, неявные или косвенные результаты.



Ошибки при задании необходимых начальных условий для отдельных программ

Большинство программ требует инициализации счетчиков, косвенных адресов, регистров, флагов и ячеек для временного хранения. Микро-ЭВМ в целом требует инициализации всех общих ячеек в ОЗУ (особо отметим косвенные адреса и счетчики).

Неправильная организация программы

К типичным ошибкам относятся:

- обход или повторение секций инициализации;
- ошибочное изменение регистров с адресами или счетчиками;
- потеря промежуточных или окончательных результатов.

Обычным источником ошибок, которые здесь не рассматриваются, является конфликт между программой пользователя и системными программами.

Простым примером такого конфликта является попытка сохранять данные программы пользователя в ячейках памяти системной программы. В этом случае всякий раз, когда выполняется системная программа, изменяются данные, которые нужны для программы пользователя.

Более сложные источники конфликтов связаны с системой прерываний, портами ввода-вывода, стеком и флагами.

Системные программы в конечном счете должны эксплуатировать те же самые ресурсы, что и программы пользователя.

При этом обычно в системных программах предусматривается сохранение и восстановление программной среды, в которой работают пользовательские программы, но это часто приводит к трудноуловимым или неожиданным последствиям.

Сделать такую операционную систему, которая была бы совершенно прозрачной для пользователя — это задача, сравнимая с выработкой правил и законов или сводов о налогах, которые не имели бы лазеек или побочных эффектов.



Распознавание ошибок Ассемблером

Большинство Ассемблеров немедленно распознает наиболее распространенные ошибки, такие как:

- **Неопределенный код операции** (обычно это неправильное написание или отсутствие двоеточия или метки);
- **Неопределенное имя** (часто это неправильное написание или отсутствие определенного имени);
- **Неверный символ** (например, **2** в двоичном числе или **B** в десятичном числе);
- **Неправильное значение** (обычно это число, которое слишком велико для 8 или 16 разрядов);
- **Отсутствует операнд**;
- **Двойное определение** (одному и тому же имени присваиваются два различных значения);
- **Недопустимая метка** (например, метка, предписанная псевдооперации, не допускающей метки);
- **Отсутствие метки** (например, при псевдооперации **EQU**, для которой требуется метка).

Эти ошибки неприятны, но они легко исправимы. Единственная трудность возникает тогда, когда ошибка (такая, как отсутствие точки с запятой у строки с комментарием) приводит Ассемблер в «**замешательство**», результатом чего является ряд бессмысленных сообщений об ошибках.

Существует, однако, много простых ошибок, которые Ассемблер не может распознать. Программисту следует иметь в виду, что его программа может содержать такие ошибки, даже если Ассемблер и не сообщил о них.

Типичны следующие примеры.

- Пропущенные строки.
- Пропущенные определения.
- Ошибки в написании, когда запись сама по себе допустима.
- Обозначение команд как комментариев.
- Если в команде, которая работает с парой регистров, задается одинарный регистр.
- Если вводится неправильная цифра, такая как **X** в десятичном или шестнадцатеричном числе или **7** в двоичном числе.

В Ассемблере могут распознаваться только такие ошибки, которые предусмотрел его разработчик.

Программисты же часто способны делать ошибки, которые разработчик не мог и вообразить, такие ошибки возможно найти при проверке программ вручную строчка за строчкой.



Распространенные ошибки в драйверах ввода-вывода

Так как большинство ошибок в драйверах ввода-вывода связано как с аппаратурным, так и с программным обеспечением, они трудно поддаются классификации. Приведем некоторые возможные случаи.

- Смешивание портов ввода и вывода.
- Попытка выполнить операции, которые физически невозможны.
- Упущенные из вида неявных эффектов аппаратуры.
- Чтение или запись без проверки состояния.
- Игнорирование различия между вводом и выводом.
- Ошибка при сохранении копии выводимых данных.

- Чтение данных до того, как они стабилизируются, или во время их изменения.
- Отсутствие изменения полярности данных, которые передаются к устройству или от устройства, работающего с отрицательной логикой.
- Смешивание действительных портов ввода-вывода с внутренними регистрами интегральных схем ввода-вывода.
- Неправильное использование двунаправленных портов.
- Отсутствие очистки состояния после выполнения команды ввода-вывода.



Распространенные ошибки в программах прерывания

Многие ошибки, связанные с прерываниями, зависят как от аппаратного, так и программного обеспечения. Самыми распространенными ошибками являются следующие.

- Отсутствие разрешения прерываний.
- Отсутствие сохранения регистров.
- Сохранение или восстановление регистров в неправильном порядке.
- Разрешение прерываний до инициализации приоритетов и других параметров системы прерываний.
- Неучет того, что реакция на прерывание включает сохранение счетчика команд в вершине стека.
- Отсутствие запрещения прерываний во время многобайтных передач или выполнения последовательности команд, которая не должна прерываться.

- Отсутствие разрешения прерываний после последовательности команд, которая должна выполняться без прерываний.
- Отсутствие очистки сигнала, вызывающего прерывание.
- Ошибка в общении с основной программой.
- Отсутствие сохранения и восстановления приоритетов.
- Отсутствие разрешения прерываний от дополнительных аппаратурных входов, которое выполняется с помощью очистки разрядов масок в регистре I.
- Неправильное использование разрядов разрешения прерываний в командах **SIM**.

Лекция 19. Введение в макроассемблер



Состав пакета

Пакет макроассемблера включает в себя основные программы, необходимые для создания, отладки и сопровождения программ на языке Ассемблера.

В состав пакета макроассемблера входят следующие программные компоненты:

- **MASM** — макроассемблер;
- **LINK** — объектный линкер;
- **SYMDEB** — символьный отладчик программ;
- **MAPSYM** — генератор символьного файла;
- **CREF** — утилита перекрестных ссылок;
- **LIB** — утилита обслуживания библиотек;
- **MAKE** — утилита сопровождения программ.

Линкер **LINK** обрабатывает выработанную MASM объектную программу с целью разрешения ссылок к другим модулям и приведения программы к виду, пригодному для загрузки в память.

Утилита **LIB** обеспечивает формирование и обслуживание библиотек объектных модулей, которые могут быть использованы **LINK** для разрешения внешних ссылок.

Отладчик **SYMDEB** реализует отладку сформированной программы на двух уровнях: на уровне символических имен и на уровне абсолютных адресов.

Программа **MAPSYM** предназначена для создания символьного файла для **SYMDEB**. Символьный файл формируется на основе информации, полученной от MASM, и необходим для символьной отладки.

Утилита **CREF** может быть использована для формирования листинга перекрестных ссылок программы, наличие которого облегчает отладку.

При помощи утилиты **MAKE** процесс разработки программ может быть автоматизирован. В файле описаний **MAKE** могут быть заданы различные алгоритмы вызовов и взаимодействия программ пакета (и не только их).

Кроме указанных программ, для создания ассемблерных исходных файлов необходим также редактор текстов, работающий в коде ASCII без управляющего кода. Многие редакторы текстов, которые обычно используют управляющие коды или другие специальные форматы в документах, обеспечивают также программирование или недокументированный режим для формирования ASCII-файлов.

Пакет макроассемблера работает в операционной системе MS-DOS или PC-DOS версии 2.0 и выше и требует наличия минимум 128K памяти (использование команды **SYMDEB** может потребовать дополнительной памяти).



Общие сведения

Макроассемблер MASM ассемблирует программы на языке Ассемблера и создает переместимые объектные файлы, которые могут редактироваться и выполняться в операционной системе MS-DOS.

Макроассемблер обеспечивает выполнение следующих функций:

- Анализ исходного текста на языке Ассемблера на предмет наличия в нем макрокоманд и/или макроопределений и обработка этих конструкций с соответствующей коррекцией исходного текста.

- Синтаксический анализ полученного текста и вывод необходимой диагностической информации.
- Формирование объектного модуля.

Воспринимая в качестве входа один файл с исходным текстом, макроассемблер может формировать до трех выходных файлов.

Файл листинга содержит распечатку исходного текста в соответствии со специфицированными директивами Ассемблера режимами и диагностическими сообщениями о результатах синтаксического анализа. Эти же сообщения дублируются на консоли.

Файл перекрестных ссылок содержит все используемые во входном тексте идентификаторы. В дальнейшем он может быть использован утилитой **CREF**.

В файле объектного кода формируется объектный модуль. Этот файл не формируется, если в тексте обнаружены ошибки.



Запуск макроассемблера

Ассемблирование исходного файла может производиться в двух режимах:

- С использованием подсказок.
- Посредством командной строки.

Для запуска макроассемблера с использованием подсказок необходимо ввести командную строку, содержащую только имя макроассемблера MASM со спецификацией подоглавления, если она требуется. MASM перейдет в диалоговый режим и серией подсказок запросит у пользователя информацию о следующих файлах (ответ заключается в наборе требуемых символов и нажатии клавиши **ENTER**):

1. **Имя исходного файла**. Если при ответе не указано расширение, предполагается **ASM**.

2. **Имя объектного файла**. Если при ответе не указано расширение, предполагается **OBJ**. Базовое имя объектного файла по умолчанию совпадает с базовым именем исходного файла.

3. **Имя файла листинга**. Если при ответе не указано расширение, предполагается **LST**. Базовое имя файла листинга по умолчанию **NUL**.

4. **Имя файла перекрестных ссылок**. Если при ответе не указано расширение, предполагается **CRF**. Базовое имя файла листинга по умолчанию **NUL**.

В конце любого ответа после символов / или — могут быть заданы опции макроассемблера, которые описаны ниже.

Если в каком-либо ответе специфицирован символ **;** MASM выйдет из диалогового режима и установит оставшиеся имена по умолчанию из следующего списка:

```
<имя исходного файла>.OBJ
NUL.LST
NUL.CRF
```

В любом ответе также могут быть заданы ответы на несколько следующих подсказок. В этом случае один ответ от другого отделяется запятой.

Для запуска MASM посредством командной строки необходимо ввести командую строку следующего вида:

```
MASM <имя исходного файла>[,[<имя объектного файла>]
[,[<имя файла листинга>][,[<имя файла перек. ссылок>]]]]
[<опции>][;]
```

Символ **;** может быть специфицирован в любом месте командной строки до того, как были определены все файлы.

В этом случае имена оставшихся неопределенными файлов принимаются по умолчанию из приведенного выше списка.

Из этого же списка принимаются по умолчанию имена файлов, спецификация которых в командной строке опущена (посредством лишней запятой).

Если в командной строке обнаружена ошибка, об этом сообщается через консоль, и MASM переходит в диалоговый режим.

Опции MASM могут располагаться в любом месте командной строки.

Следующие базовые имена выходных файлов MASM имеют фиксированный смысл (независимо от того, как запускается MASM):

- **NUL** — соответствующий файл не формируется;
- **PRN** — соответствующий файл направляется на печать.

Имя каждого файла может сопровождаться информацией о подоглавлении, содержащем этот файл, иначе поиск исходного файла или создание выходного файла будет осуществляться в текущем подоглавлении.

Работа MASM может быть в любой момент прекращена нажатием клавиш **CTRL-C**.



Опции MASM

Опции MASM позволяют в некоторой степени управлять работой макроассемблера вне связи с исходной программой.

Каждая опция обозначается предшествующим символом / или — и может кодироваться как строчными, так и заглавными буквами.

Опции могут располагаться в любом месте командной строки или ответа на подсказку.

Ниже приведен список опций MASM с описанием выполняемых ими функций.

/A

Сегменты в объектном файле располагаются в алфавитном порядке. При отсутствии опции расположение сегментов соответствует порядку в исходном файле.

/S

Сегменты в объектном файле располагаются в порядке следования в исходном файле. Эта опция введена для совместимости с XENIX.

/B<число>

Установить размер буфера исходного файла (в килобайтах). Увеличение размера буфера ускоряет ассемблирование, но требует больше памяти.

Размер буфера может варьироваться от 1 до 63 (K). Если опция не задана, полагается 32 (32K).

/D

Диагностические сообщения после 1-го прохода поместить в листинг программы. Многие ошибки 1-го прохода исправляются на 2-м проходе, и, если не задано **/D**, в листинг не попадают. Задание этой опции дает более глубокую диагностику исходного текста. При спецификации **/D** ошибки как 1-го, так и 2-го проходов выдаются на консоль, даже если файл листинга не создается.

/D<символ>

Определить символ. Указанный символ вводится в исходный текст как пустая строка (аналогично использованию директивы **EQU**) и может быть использован в директивах условного ассемблирования.

/I<путь>

Задание пути поиска файлов, подключаемых в исходный текст директивой **INCLUDE** без явного указания пути. Указание пути в **INCLUDE** более приоритетно, чем в опции **/I**.

/ML

Установить различие между строчными и заглавными буквами в метках, переменных и именах. При отсутствии этой опции строчные буквы автоматически преобразуются в заглавные. Опция может потребоваться для совместимости с программами на регистро-чувствительных языках.

/MX

Установить различие между строчными и заглавными буквами в общих и внешних переменных. Опция подобна **/ML**, но ее действие распространяется лишь на имена, используемые в директивах **PUBLIC** или **EXTRN**.

/MU

Преобразовать в общих и внешних именах строчные буквы в заглавные. Опция включена по умолчанию и введена для совместимости с XENIX.

/N

Запретить вывод в файл листинга таблиц, макроструктур, записей, сегментов и имен. На генерируемый код опция не влияет.

/P

Контроль запрещенного кода. Выполнение некоторых инструкций может привести к нежелательным последствиям (например, загрузка регистра **CS**). Кодирование таких инструкций может быть запрещено опцией **/P**, наличие которой в таких случаях вызывает генерацию ошибки с кодом 100. Директива .286p отменяет эту опцию и разрешает кодирование запрещенных инструкций.

/R

Генерация кода для процессора с плавающей точкой. Генерируются коды инструкций арифметики с плавающей точкой, которые могут быть выполнены только при наличии сопроцессоров.

/E

Генерация кода для эмуляции плавающей точки.

Генерируется код, эмулирующий инструкции арифметики с плавающей точкой. Эта возможность используется при отсутствии указанных сопроцессоров.

При использовании этого режима необходимо наличие специальной библиотеки эмуляции, содержащей модули, моделирующие операции с плавающей точкой. Эта библиотека эмуляции должна использоваться при обработке объектного модуля с помощью **LINK**.

/V

Включить в диагностику на консоль информацию о числе обработанных строк и символов. При отсутствии этой опции на консоль выдается информация об ошибках и памяти.

/X

Выводить в листинг тела блоков **IF** (IF, IFE, IF1, IF2, IFDEF, IFNDEF, IFB, IFNB, IFIDN и IFDIF), для которых условия ассемблирования оказываются ложными и код по этой причине не генерируется. Следующие директивы Ассемблера влияют на действие опции **/X**:

- **.SFCOND** подавляет печать «ложных» блоков;
- **.LFCOND** — разрешает печать «ложных» блоков;
- **.TFCOND** — каждая обработка директивы меняет состояние опции на противоположное.

/Z

Выводить на консоль строки исходного файла, содержащие ошибки. При отсутствии этой опции на консоль выдаются только сообщение об ошибке и номер строки. Кодирование опции замедляет работу макроассемблера.

/C

Создать файл перекрестных ссылок. Файл создается, даже если он подавлен командной строкой или ответом на подсказку. В последнем случае имя файла устанавливается по умолчанию (**<имя исходного файла>.CRF**). Опция **/C** введена для совместимости с XENIX.

/L

Аналогично **/C**, но относится к файлу листинга (с учетом умалчиваемого имени файла).

/T

Подавить все сообщения, если в исходном тексте не встретится ошибок.



LINK: линкер модулей

Объектный линкер предназначен для создания исполнительных файлов из объектных файлов, сформированных MASM или компиляторами C или PASCAL.LINK формирует переместимый исполнительный код, снабженный информацией перемещения, используя которую, MS-DOS сможет загрузить в память и исполнить соответствующую программу. LINK может формировать программы, содержащие свыше 1Мб кода и данных. Воспринимая в качестве входа 2 файла, LINK может формировать 2 выходных файла.

```
имя.LIB                         имя.MAP
(библиотека)  \               /  (план)
                    LINK
имя.OBJ       /               \  имя.EXE
(объектный файл)                (исполнительный файл)
```

Расширения имен файлов, показанные на схеме принимаются по умолчанию.

Объектный файл содержит объектные модули программных сегментов, сформированные MASM или компилятором языка высокого уровня.

Библиотеки содержат наборы модулей, на которых могут ссылаться программные сегменты в объектном файле. Библиотечные файлы формируются при помощи утилиты **LIB**.

Основным результатом работы LINK является исполнительный файл, содержащий программу в виде, пригодном для загрузки в память и исполнения.

Файл плана является необязательным и содержит, если он формируется, некоторую диагностическую и служебную информацию, которая затем при посредстве утилиты **MAPSYM** может быть использована в процессе отладки программы.

Файл плана содержит имена, загрузочные адреса и длины всех сегментов программы. Кроме того, сюда входят имена и загрузочные адреса групп в программе, адрес точки входа, а также сообщения о возможных ошибках.

Если задана опция **/MAP**, в файл включаются имена общих символов и их загрузочные адреса.

Если заданы опции **/HIGH** или **/DSALLOCATE** и объем программы и данных в совокупности не превышает 64K, план может содержать символы с необычно большими адресами сегментов. Эти адреса отражают переменные, расположенные ниже действительного начала сегмента. Пример:

```
FFF0:0A20   TEMP
Адрес TEMP - 00:920h.
```

Необходимо иметь ввиду, что, кроме двух выходных файлов, LINK может формировать временный файл с именем **VM.TMP**. Это происходит в том случае, когда линкеру не хватает оперативной памяти. Создание файла **VM.TMP** сопровождается сообщением на консоли и всегда осуществляется в текущем подоглавлении. В этом случае нельзя использовать опцию **/PAUSE** и снимать дискету, если она находится на активном драйве, до того, как LINK не уничтожит файл **VM.TMP**. Не рекомендуется создавать в текущем подоглавлении файл с таким именем, который в этом случае может быть испорчен.

Запуск LINK

Запуск LINK может быть осуществлен тремя способами:

- С использованием подсказок.
- При помощи командной строки DOS.
- С использованием файла ответа.

Для запуска LINK с использованием подсказок необходимо ввести командную строку, содержащую только имя линкера LINK со спецификацией подоглавления, если она требуется. LINK перейдет в диалоговый режим и серией подсказок запросит у пользователя информацию о следующих файлах (ответ заключается в наборе требуемых символов и нажатии клавиши **ENTER**):

1. **Имя объектного файла**. Если при ответе не указано расширение, предполагается **OBJ**. Если нужно определить несколько файлов, их имена разделяются символом **+**. Если все имена не помещаются на одной строке, ввод имен можно продолжить, поставив символ **+** в последнюю позицию текущей строки. В этом случае LINK повторит запрос для ввода дополнительных имен.

2. **Имя исполнительного файла**. Если при ответе не указано расширение, предполагается **EXE**. Базовое имя исполнительного файла по умолчанию совпадает с базовым именем объектного файла.

3. **Имя файла плана модуля**. Если при ответе не указано расширение, предполагается **MAP**. Базовое имя по умолчанию **NUL**.

4. **Имя библиотеки**. Если при ответе не указано расширение, предполагается **LIB**. Можно задавать несколько имен библиотек по аналогии с OBJ-файлами. Если, не вводя имени, сразу нажать **ENTER**, библиотеки использоваться не будут.

Если в каком-либо ответе специфицирован символ **;** LINK выйдет из диалогового режима и установит оставшиеся имена по умолчанию из следующего списка:

```
<имя объектного файла>.EXE
NUL.MAP
Библиотеки не используются.
```

В любом ответе также могут быть заданы ответы на несколько следующих подсказок. В этом случае один ответ от другого отделяется запятой.

Для запуска LINK посредством командной строки, необходимо ввести командую строку следующего вида:

```
LINK <имя объектного файла>[,[<имя исполнительного файла>]
[,[<имя файла плана>][,[<имя библиотеки>]]]] [<опции>][;]
```

Символ **;** может быть специфицирован в любом месте командной строки до того, как были определены все файлы. В этом случае имена оставшихся неопределенными файлов принимаются по умолчанию из приведенного выше списка. Из этого же списка принимаются по умолчанию имена файлов, спецификация которых в командной строке опущена (посредством лишней запятой). Если в командной строке обнаружена ошибка, об этом сообщается через консоль, и LINK переходит в диалоговый режим.

Если специфицирована хотя бы одна из опций **/MAP** или **/LINENUMBERS**, файл плана создается независимо от того, указано ли его имя в командной строке. В этом случае, если его имя не специфицировано, оно принимается по умолчанию — **<имя объектного файла>.MAP**.

При указании нескольких объектных файлов или библиотек их имена разделяются символами **+**.

Если определены не все файлы (но не опущены посредством лишней запятой, и не специфицирована установка оставшихся имен по умолчанию указанием символа **;**), LINK входит в диалоговый режим и запрашивает оставшиеся неопределенными имена через подсказки.

Спецификации имен файлов и опции могут быть заранее занесены в специальный файл ответа. Имя этого файла с предшествующим символом **@** и указанием пути поиска, если он нужен, может быть помещено в любом месте ответа на подсказку или командной строки и трактуется, как если бы содержимое файла ответа было непосредственно вставлено в это место. Следует, однако, помнить, что комбинация символов

CARRIAGE-RETURN / LINE-FEED в файле ответа интерпретируется как **ENTER** в подсказке или запятая в командном файле.

Общий вид файла ответа:

```
<имя объектного файла>
[<имя исполнительного файла>]
[<имя файла плана>]
[<имя библиотеки>]
```

Каждая группа файлов должна задаваться на отдельной строке, а файлы в группе, если их несколько, должны разделяться символом **+**. Если группа не помещается на одной строке, в последней позиции строки должен стоять признак продолжения — символ **+**. В любой строке файла ответа после символа / могут быть заданы опции LINK.

В файле ответа могут быть опущены компоненты, уже определенные ответами на подсказки или командной строкой.

При обнаружении в файле ответа символа **;** остаток файла игнорируется, и оставшиеся неопределенными имена устанавливаются по умолчанию из приведенного выше списка.

При использовании файла ответа его содержимое выдается на консоль в форме подсказок. Если определены не все имена, LINK переходит в диалоговый режим.

Если файл ответа не содержит комбинации символов

CARRIAGE-RETURN / LINE-FEED или символа **;** LINK выдает на консоль последнюю строку файла и ожидает нажатия **ENTER**.

Имя каждого файла может сопровождаться информацией о подоглавлении, содержащем этот файл, иначе поиск исходного файла или создание выходного файла будет осуществляться в текущем подоглавлении.

Работа LINK может быть в любой момент прекращена нажатием клавиш **CTRL-C**.

Опции LINK

Все опции LINK обозначаются предшествующим символом / и могут быть сокращены произвольным образом, но так, чтобы код оставался уникальным среди опций.

Ниже приведены описания всех опций LINK (в скобках указаны минимальные сокращения):

/HELP (HE)

Выдать список действующих опций. Эту опцию нельзя использовать вместе с именем файла.

/PAUSE (P)

Пауза перед записью модуля в **EXE-файл** (и после записи в MAP-файл, если это предусмотрено). Во время этой паузы можно при необходимости переставить дискеты. Если используется файл **VM.TMP**, он должен находиться на той же дискете, что и EXE-файл.

/EXEPACK (E)

Установить компактную запись последовательностей одинаковых бит. Такой EXE-файл имеет меньший объем и быстрее загружается в па-

мять, но его нельзя отлаживать при помощи **SYMDEB**. Опция дает эффект, если программа содержит длинные потоки идентичных битов и требует большого числа (более 500) перемещений при загрузке.

/MAP (M)

Формировать MAP-файл. Файл формируется, даже если он не специфицирован при запуске LINK, и имеет в этом случае умалчиваемое имя.

/LINENUMBERS (LI)

Зафиксировать в MAP-файле номера строк исходного файла. Эта информация может в дальнейшем использоваться **MAPSYM** и **SYMDEB**. Запись номеров строк будет производиться, если создается MAP-файл и объектный модуль содержит данные о строках исходного текста. Компиляторы FORTRAN и PASCAL (версии 3.0 и выше) и C (версии 2.0 и выше) такие данные автоматически формируют; в MASM это не предус мотрено. Если MAP-файл не специфицирован, его можно создать принудительно, указав описываемую опцию в подсказке на этот файл.

/NOIGNORECASE (NOI)

Установить различие между строчными и заглавными буквами. Различие может быть установлено также опциями **/ML** и **/MX MASM**.

/NODEFAULTLIBRARYSEARCH

Игнорировать умалчиваемые (NOD) библиотеки, ссылки на которые содержатся в объектном модуле (их туда помещают компиляторы языков высокого уровня). Используются только библиотеки, специфицированные при запуске LINK.

/STACK:<число> (ST)

Установить размер стека (в байтах). Информация о размере стека, содержащаяся в объектном модуле, игнорируется. Размер стека может быть задан в виде десятичного, 8-ричного (с предшествующим 0) или 16-ричного (с предшествующими 0 и x на малом регистре) числа в пределах от 1 до 65535. Размер стека может быть изменен утилитой **EXEMOD**.

/CPARMAXALLOC:<число>

Установить максимальное число (C) 16-байтных параграфов, необходимых при загрузке программы в память. Обычно LINK устанавливает максимальное число параграфов — 65535. Указание этой опции позволяет более эффективно использовать память. Число параграфов может

быть задано в виде десятичного, 8-ричного (с предшествующим 0) или 16-ричного (с предшествующими 0 и x на малом регистре) числа в пределах от 1 до 65535. Если число параграфов недостаточно для размещения программы, LINK наращивает его до минимально подходящего. Число параграфов может быть изменено утилитой **EXEMOD**. Кроме размещения программы, опция может понадобиться для команды **! SYMDEB**.

/HIGH (H)

Установить адрес начала программы на наивысший возможный адрес свободной памяти. Без этой опции установка осуществляется на минимальный возможный адрес.

/DSALLOCATE (D)

Обработать группу с именем **DGROUP**. Обычно LINK присваивает младшему байту группы смещение 0000h. При задании этой опции старшему байту группы с именем **DGROUP** присваивается смещение FFFFh. В результате данные будут размещаться в областях программы с максимально большими адресами. Опция **/D** обычно применяется вместе с опцией **/H** для более эффективного использования незанятой памяти до старта программы. LINK предполагает, что все свободные байты в **DGROUP** занимают память непосредственно перед программой. Для

использования группы необходимо загрузить в регистр сегмента адрес начала **DGROUP**.

/NOGROUPASSOCIATION

Игнорировать группы (**NOG**) при присвоении адресов элементам данных и кода. Опция введена для совместимости с ранними версиями компиляторов FORTRAN и PASCAL (версии MICROSOFT 3.13 и ранее и IBM до 2.0). Не рекомендуется использовать эту опцию в других целях.

/OVERLAYINTERRUPT:<число>

Установить номер прерывания (O) при загрузке оверлейного модуля.

Указанное число замещает номер стандартного оверлейного прерывания (03Fh). Номер может быть задан в виде десятичного, 8-ричного (с предшествующим 0) или 16-ричного (с предшествующими 0 и x на малом регистре) числа в пределах от 0 до 255. MASM не способствует созданию оверлейных программ. Поэтому только при помощи опции **/O** ассемблерные модули могут быть включены в оверлейные программы на языках высокого уровня, компиляторы которых поддерживают оверлей.

Не рекомендуется устанавливать номер, совпадающий с каким-либо другим прерыванием.

/SEGMENTS:<число> (SE)

Установить максимальное число сегментов, которое может обработать LINK. Число может быть задано в десятичной, 8-ричной (с предшествующим 0) или 16-ричной (с предшествующими 0 и x на малом регистре) форме в пределах от 1 до 1024. При отсутствии спецификации опции полагается 128. Память выделяется с учетом этого максимального числа сегментов.

/DOSSEG (DO)

Упорядочить сегменты в EXE-файле. При спецификации этой опции сегменты располагаются в следующей последовательности:

- сегменты с классом **CODE**;
- другие сегменты, не входящие в группу **DGROUP**;
- сегменты, входящие в **DGROUP**.

Особенности работы LINK

LINK создает исполнительный файл путем конкатенации кода программы и сегментов данных, соответствующих корректным инструкциям исходного текста. Эта сцепленная форма сегментов и является тем «исполнительным представлением», которое непосредственно копируется в память при загрузке программы.

Частично управлять редактированием программных сегментов можно заданием атрибутов в директиве **SEGMENT** или использованием директивы **DGROUP** для формирования группы сегментов. Эти директивы определяют целую группу ассоциаций, классов и типов выравнивания, а также определяют порядок и относительные начальные адреса сегментов программы. Эта информация является дополнительной к той, которая задается опциями LINK.

- Выравнивание сегментов

Для установки начального адреса сегмента LINK использует задаваемый директивой **SEGMENT** тип выравнивания: **BYTE**, **WORD**, **PARA** или **PAGE**. Эти ключевые слова обеспечивают выравнивание начала сегмента соответственно по границе байта, слова (2 байта), параграфа (16 байтов) или страницы (256 байтов). По умолчанию используется тип **PARA**.

Байты, пропускаемые из-за выравнивания, заполняются двоичными нулями.

- Номер кадра

Вычисляемый LINK начальный адрес сегмента зависит от типа выравнивания сегмента и размеров уже скопированных в исполнительный файл сегментов.

Этот адрес состоит из смещения и канонического номера кадра. Канонический адрес кадра определяет адрес первого параграфа в памяти, содержащего один или более байтов сегмента. Номер кадра всегда кратен 16. Смещением является расстояние в байтах от начала параграфа до первого байта сегмента.

Для типов **PAGE** и **PARA** смещение всегда нулевое, а для типов **BYTE** и **WORD** может быть ненулевым.

Номер кадра может быть получен из MAP-файла. Его содержат первые 5 16-ричных цифр start-адреса сегмента.

- Последовательность сегментов

LINK копирует сегменты в исполнительный файл в той же последовательности, в какой он их считывает из объектных файлов.

Сегменты, имеющие идентичные имена классов, считаются принадлежащими к одному типу классов и копируются в исполнительный файл как непрерывный блок.

- Комбинированные сегменты

Для определения того, будут ли два или более сегмента, имеющие одно и то же имя, соединены в один большой сегмент, LINK использует комбинации типов сегментов. В языке Ассемблера имеются следующие типы комбинаций: **PUBLIC**, **STACK**, **COMMON**, **MEMORY**, **AT** и **PRIVATE**.

Если сегмент имеет тип комбинации **PUBLIC**, LINK автоматически соединяет его с другими сегментами, имеющими то же имя и принадлежащими к тому же классу. При соединении сегментов предполагается, что сегменты непрерывны и все адреса в сегментах доступны через смещение относительно адреса кадра. Результат получается таким же, как если бы полученный большой сегмент был определен в исходном файле сплошным куском.

LINK сохраняет тип выравнивания каждого сегмента. Это означает, что, хотя сегменты и включены в один большой сегмент, код и данные сегментов сохраняют свои типы выравнивания.

Если размеры соединяемых сегментов превышают 64K, выдается сообщение об ошибке.

Если сегмент имеет тип комбинации **STACK**, LINK выполняет ту же операцию, что и в случае **PUBLIC**. Различие заключается в том, что для **STACK**-сегментов в исполнительный файл записывается начальное значение указателя стека, которое представляет собой смещение от конца первого по порядку сегмента стека или комбинированного сегмента стека. В этом случае при использовании типа **STACK** для сегментов стека программисту нет необходимости предусматривать в программе загрузку регистра **SS**.

Если сегмент имеет тип комбинации **COMMON**, LINK автоматически соединяет его с другими сегментами, имеющими то же имя и принадлежащими к тому же классу. Однако, когда LINK соединяет общие сегменты, начало каждого сегмента устанавливается на один адрес, в результате чего образуются серии перекрывающихся сегментов. В итоге получается один сегмент, который по длине не превышает самый длинный из комбинируемых сегментов.

Сегменты с типом комбинации **MEMORY** трактуется LINK в точности так же, как и **PUBLIC**-сегменты. MASM обеспечивает тип **MEMORY** для совместимости с линкерами, выделяющие **MEMORY** как особый тип комбинации.

Сегмент имеет тип комбинации **PRIVATE** в том случае, когда в исходном файле нет точных указаний относительно его типа комбинации. LINK не объединяет **PRIVATE**-сегменты.

- Группы

Объединение нескольких сегментов в группу позволяет адресовать их относительно одного адреса кадра. При этом неважно, принадлежат ли эти сегменты к одному классу. Когда LINK обнаруживает группу, он соответствующим образом перестраивает все адресные ссылки в ней.

Сегменты в группе не являются смежными, не принадлежат к одному классу и имеют разные типы комбинации. Но суммарный объем всех сегментов в группе не должен превышать 64K.

Группы не влияют на порядок загрузки сегментов в память. Даже если используются имена классов и объектные файлы вводятся в соответствующей последовательности, нет гарантии, что сегменты будут смежными. На практике LINK может поместить не принадлежащий группе сегмент в те же 64K памяти.

Хотя в LINK и нет строгой проверки того, помещаются ли все сегменты группы в 64K памяти, при обнаружении нарушения этого условия будет выдано сообщение о переполнении согласования.

- Согласования

Когда в процессе работы LINK уже известны адреса всех сегментов программы и организованы все комбинации сегментов и группы, линкер имеет возможность «согласовать» некоторые неразрешенные ссылки к меткам и переменным. Для этого LINK вычисляет соответствующие адрес сегмента и смещение и замещает временные значения, сгенерированные Ассемблером, на новые значения.

В соответствии с типами ссылок LINK реализует следующие типы согласований:

- Короткие.
- Внутренние относительно себя.
- Внутренние относительно сегмента.
- Длинные.

Размер вычисляемого значения зависит от типа ссылки. Если LINK обнаруживает ошибку в предсказанном размере ссылки, выдается сообщение о переполнении согласования. Это может произойти, например, когда программа пытается использовать 16-битовое смещение для доступа к инструкции в сегменте, имеющем другой адрес кадра. Это же сообщение может быть выдано, если все сегменты в группе не помещаются внутри блока памяти в 64K.

Короткая ссылка имеет место в инструкции **JMP**, передающей управление на помеченную инструкцию в том же сегменте или группе, отстоящую от **JMP** не более, чем на 128 байтов. Для такой ссылки LINK вырабатывает 8-битовое число со знаком. Если инструкция, на которую передается управление, находится в другом сегменте или группе, то есть, имеет другой адрес кадра, или отстоит более, чем на 128 байтов в любом направлении, формируется сообщение об ошибке.

Внутренняя относительно себя ссылка имеет место в инструкциях, адресующих данные относительно того же сегмента или группы. Для такой ссылки LINK формирует 16-битовое смещение. Если данные не принадлежат тому же сегменту или группе, выдается сообщение об ошибке.

Внутренняя относительно сегмента ссылка имеет место в инструкциях, адресующих данные в определенном сегменте или группе или относительно указанного регистра сегмента. Для этой ссылки LINK вы-

рабатывает 16-битовое смещение. Если это смещение внутри специфицированного кадра оказывается больше 64K или меньше 0 или если начало канонического кадра, содержащего требуемые данные, неадресуемо, выдается сообщение об ошибке.

Длинная ссылка имеет место в инструкциях **CALL**, передающих управление в другой сегмент или группу. LINK в этом случае вырабатывает 16-битовый адрес кадра и 16-битовое смещение. Если вычисленное смещение больше 64K или меньше 0 или если начало канонического кадра, в который передается управление, неадресуемо, формируется сообщение об ошибке.

- Поиск библиотек

Процедура поиска библиотеки, иногда требуемой для разрешения внешних ссылок, обладает некоторыми особенностями. Если путь поиска указан вместе с именем библиотеки в командной строке, поиск осуществляется только там. Если же путь явно не указан, поиск производится в следующей последовательности:

1. В текущем подоглавлении.

2. Если в командной строке заданы один или несколько путей поиска для других библиотек, LINK просматривает их в порядке следования в строке.

3. На путях, определенных переменной **LIB** команды **DOS SET**. При помощи команды **SET** могут быть заданы несколько путей поиска, разделяемых точкой с запятой. Вид команды SET:

```
SET LIB=<список путей>
```



SYMDEB: символьный отладчик программ

При помощи символьного отладчика SYMDEB могут быть выполнены следующие функции:

- Просмотр и исполнение кода программы.
- Внесение в тело программы точек выхода, которые останавливают исполнение программы.

- Проверка и изменение в памяти значений переменных.
- Ассемблирование и реассемблирование кода.
- Отладка программ, использующих соглашения языков Microsoft об эмуляции арифметики с плавающей точкой.

Имеется возможность временного выхода в DOS с последующим возвратом в SYMDEB и сохранением его состояния.

SYMDEB имеет несколько способов адресации данных и инструкций в памяти. К различным фрагментам программы можно получить доступ через адреса, глобальные символы или номера строк, что упрощает размещение и отладку специфических участков кода.

Программы на языках C, PASCAL и FORTRAN могут быть отлажены как на уровне исходных файлов, так и на уровне исполнительного кода. При этом пользователю доступны исходные предложения, реассемблированный машинный код или их комбинация в зависимости от режима работы SYMDEB.

Подготовка символьной отладки

Сущность символьной отладки заключается в том, что в процессе отладки программы имеется возможность ссылаться на элементы исходной программы (очевидно, что для этого исходный файл должен быть доступен SYMDEB).

SYMDEB является сильным отладочным средством даже без своих возможностей символьной отладки, однако, при этом в значительной степени теряется наглядность работы, что часто существенно усложняет отладку. С другой стороны, заметно упрощается процесс подготовки к отладке, особенно для программ на языке Ассемблера.

Для символьной отладки при помощи SYMDEB необходимо предварительно сформировать специальный символьный файл, куда должна быть занесена информация, позволяющая «привязать» элементы исходной программы (переменные, метки, номера строк) к относительным адресам внутри программных сегментов на уровне исполнительного кода.

Необходимо помнить, что все имена, участвующие в символьной отладке, должны быть объявлены (средствами конкретного языка) как общие, то есть, должны быть доступными программе LINK при согласовании внешних ссылок.

Исходная информация для символьной отладки

Этапы формирования символьного файла существенно зависят от того, каким транслятором обрабатывалась исходная программа. Дело в том, что некоторые компиляторы не обеспечивают информацию о номерах строк исходного файла, и для такой программы допустима лишь ограниченная символьная отладка (без адресации по номерам строк). Кроме того, при работе с макроассемблерами имеется целый ряд характерных для языков типа Ассемблера особенностей, вносящих в процесс отладки определенную специфику. Сущность этой специфики станет понятной при ознакомлении с командами SYMDEB.

По указанным причинам при работе с SYMDEB и особенно при подготовке символьной отладки следует всегда учитывать, каким компилятором обрабатывалась исходная программа.

SYMDEB совместим со следующими компиляторами:

- MICROSOFT FORTRAN версии 3.0 и выше
- MICROSOFT PASCAL версии 3.0 и выше
- MICROSOFT C версии 2.0 и выше
- MICROSOFT макроассемблер версии 1.0 и выше
- MICROSOFT BASIC COMPILER версии 1.0 и выше
- MICROSOFT BUSINESS BASIC COMPILER версии 1.0 и выше
- IBM PC FORTRAN версии 2.0 и выше
- IBM PC PASCAL версии 2.0 и выше
- IBM PC макроассемблер версии 1.0 и выше
- IBM PC BASIC COMPILER версии 1.0 и выше

Из них лишь следующие компиляторы поддерживают работу SYMDEB на уровне номеров строк исходного файла:

- MICROSOFT FORTRAN версии 3.0 и выше
- MICROSOFT PASCAL версии 3.0 и выше
- MICROSOFT C версии 2.0 и выше
- IBM PC FORTRAN версии 2.0 и выше
- IBM PC PASCAL версии 2.0 и выше

Конечной целью подготовки символьной отладки является создание символьного файла. В общем случае это осуществляется путем обработки исходной программы соответствующим компилятором и программой LINK и формирования на основе полученной информации собственно символьного файла при помощи программы **MAPSYM**.

Для создания символьного файла при работе с макроассемблерами необходимо выполнить следующие шаги:

1. Символы, которые будут использованы SYMDEB, объявить как общие. Среди этих символов могут быть имена процедур, переменных и меток. Имена сегментов и групп не могут быть объявлены общими, но они автоматически включаются LINK в MAP-файл и могут быть использованы при отладке. Пользователь может объявить фиктивные метки, которые в программе не используются, но могут пригодиться при расстановке точек выхода.

2. Ассемблировать исходный файл макроассемблером.

3. Обработать полученный объектный файл программой LINK с опцией **/MAP** и получить EXE- и MAP-файлы.

4. Использовать MAPSYM для создания символьного файла.

Для создания символьного файла при работе с другими совместимыми с SYMDEB компиляторами (кроме макроассемблеров) необходимо выполнить следующие шаги:

1. Компилировать исходный файл. При этом для упрощения отладки желательно не использовать имеющиеся в компиляторе средства оптимизации. При необходимости (и возможности) следует обеспечить наличие в объектном файле информации о номерах строк исходной программы.

2. Полученный объектный файл обработать программой LINK с опциями **/MAP** и, если нужно, **/LINENUMBERS** и получить EXE- и MAP-файлы.

3. Использовать MAPSYM для создания символьного файла.

4. Запустить SYMDEB для символьной отладки.

5. Использовать команду SYMDEB **Go (G)** для запуска тестируемой программы с первой процедуры или функции. Это обеспечивается специальной программой запуска из библиотеки программ данного языка. Обычно пользователю не нужна трассировка этой программы, и он желает начинать отладку с момента начала работы своей программы. В C-программе первая выполняемая функция всегда называется **_main**

(компилятор С добавляет символ _ к имени процедуры **main**), в FORTRAN-программе — main, в PASCAL — первая процедура в исходном файле).

Формирование символьного файла при помощи MAPSYM

Программа **MAPSYM** предназначена для преобразования информации из MAP-файла, сформированного LINK, в форму, понятную SYMDEB. Создаваемый символьный файл может содержать до 10000 символов для каждого сегмента и столько сегментов, сколько допустимо с точки зрения машинной памяти.

Запуск **MAPSYM** обеспечивается введением следующей командной строки:

```
MAPSYM [/L | -L]<имя MAP-файла>
```

Имя MAP-файла может содержать спецификацию пути поиска в структуре оглавления, если это необходимо. Если не указано расширение имени, предполагается MAP.

Задание единственной опции **/L** (или **-L**) позволяет получить на экране дисплея информацию о преобразовании, куда входят имена определенных в программе групп, адрес начала программы, число сегментов и число символов в каждом сегменте.

Символьный файл имеет базовое имя, совпадающее с базовым именем MAP-файла, и расширение **SYM** и всегда создается в текущем подоглавлении.

Запуск SYMDEB

Запуск SYMDEB обеспечивается введением следующей командной строки:

```
SYMDEB [<опции>][<символьные файлы>] [<исполнительный файл>]
[<аргументы>]
```

В некоторых случаях (например, при использовании оверлея или при работе с драйверами) может потребоваться несколько символьных файлов. Все они в командной строке должны быть специфицированы до имени исполнительного файла, так как имена, стоящие после имени исполнительного файла считаются аргументами.

Исполнительным файлом может быть двоичный или EXE-файл, содержащий тестируемую программу. SYMDEB загружает этот файл в память.

Аргументы, если они указаны, передаются SYMDEB в заголовок тестируемой программы в точности, как они специфицированы, и могут быть использованы этой программой как параметры.

Стартовав, SYMDEB выдает сообщение об этом и символ **-**, после которого можно вводить команды SYMDEB.

Запуск SYMDEB для символьной отладки

При использовании символьной отладки в командной строке запуска SYMDEB должен быть специфицирован символьный файл.

При загрузке нескольких символьных файлов сразу будет открыт только один из них. Если один из специфицированных символьных файлов имеет базовое имя, совпадающее с базовым именем исполнительного файла, открывается именно он, иначе будет открыт первый из указанных символьных файлов. Во время сеанса SYMDEB для открытия конкретного файла может быть использована команда **XO**. Так как одновременно может быть открыт только один символьный файл, предыдущий файл при выполнении команды **XO** закрывается.

Исполнительный файл может быть специфицирован при загрузке символов. Символы могут быть загружены без исполнительного файла (это может потребоваться для отладки резидентной программы). Если возникнет необходимость в загрузке исполнительного файла позже в текущем сеансе, могут быть использованы команды **N** или **L**.

Не следует переименовывать символьные файлы, так как тогда при загрузке они будут иметь неверные адреса.

Запуск SYMDEB только с исполнительным файлом

Если пользователь не нуждается в символьном файле или не имеет исходного файла программы, в командной строке запуска SYMDEB он может опустить имя символьного файла.

SYMDEB будет загружать исполнительные файлы, имеющие расширения имен **EXE**, **BIN**, **HEX** или **COM**.

Всякий раз при загрузке исполнительного файла SYMDEB формирует 256-байтный заголовок в сегменте памяти с наименьшим возможным адресом и копирует содержимое файла непосредственно после заголовка. SYMDEB помещает размер программы в байтах в пару регистров **BX:CX** и устанавливает значения всех остальных регистров в соответствии с содержимым файла.

Для **EXE-** и **HEX**-файлов заголовок при загрузке будет разорван на части. Поэтому размер программы в этом случае не соответствует размеру файла, как это имеет место для **COM-** и **BIN**-файлов.

Запуск SYMDEB без файлов

Если в командной строке запуска не указаны имена файлов, SYMDEB создает заголовок программы, но ничего не загружает. Для продолжения работы пользователь может использовать команды ассемблирования (**A**), ввода (**E**), установки имени (**N**) или загрузки (**L**).

При запуске без файлов SYMDEB перед началом отладки очищает флаги, в регистр **IP** загружает значение 0100h, устанавливает регистры сегментов на «дно» свободной памяти и обнуляет остальные регистры.

Опции SYMDEB

Опции SYMDEB обозначаются предшествующими символами / или **-** и могут кодироваться как заглавными, так и строчными буквами. В командной строке запуска SYMDEB они располагаются перед именем исполнительного файла.

Имена файлов, содержащие символ **-**, во избежание путаницы должны быть изменены.

Описание опций SYMDEB приведено ниже.

/IBM (или I)

Установить совместимость с IBM. При работе на не IBM-машине эта опция позволяет учитывать некоторые особенности аппаратуры IBM (контроллер прерываний 8259, дисплей IBM и другие особенности BIOS). При работе на IBM-машине эти особенности распознаются автоматически.

/K

Разрешение интерактивного выхода по ключу.

При задании этой опции клавиша **SCROLL-LOCK** (**BREAK**) программируется таким образом, что ее нажатие останавливает исполнение программы. Это может понадобиться, например, для выхода из бесконечного цикла.

Интерактивный ключ работает примерно также, как ключ прерывания от аппаратуры, но менее надежно. В некоторых ситуациях (например, когда замаскированы прерывания) он не срабатывает. Если программа ожидает ввода, предпочтительней использовать **CTRL-C**, чем

BREAK. В IBM PC AT в тех же целях, но даже без опции **/K**, может быть использован ключ **SYS REQ**.

/N

Разрешение выхода по немаскируемому прерыванию. Для работы с немаскируемыми прерываниями вычислительная система должна иметь соответствующее аппаратное обеспечение. Опцию **/N** можно использовать со следующими продуктами:

- специальная утилита отладки фирмы IBM
- зонд математического обеспечения фирмы ATRON

При использовании опции **/N** SYMDEB требуется аппаратура, поставляемая с этими продуктами. Дополнительного математического обеспечения не требуется. При использовании одного из этих продуктов в системе, отличной от IBM, для обеспечения выхода нужно использовать опцию **/N**. Но эту опцию нельзя кодировать при работе на IBM PC. Использование системы выхода по немаскируемому прерыванию предпочтительней интерактивного выхода по ключу, так как не зависит от состояний прерываний и других условий.

/S

Разрешение смены экрана. Эта опция позволяет разделить экраны: один для отладчика, другой — для отлаживаемой программы. Это удобно, например, при отладке графических программ, но требует дополнительно 32K памяти. Опция **/S** работает только на IBM-машинах и некоторых совместимых с ними компьютерах. При работе на совместимых компьютерах в этом случае должна быть специфицирована также и опция **/IBM**. Опция **/S** не может быть использована с графикой, требующей более 32K памяти.

/"команды"

Выполнить серию команд SYMDEB. Команды должны разделяться символом ; и выполняются после загрузки файлов, но перед их исполнением. Этой опцией могут быть, например, заданы команды, выполняемые в начале каждого сеанса SYMDEB.

Команды SYMDEB

При помощи команд SYMDEB реализуется алгоритм отладки, а также обеспечивается выполнение основных функций SYMDEB. Команды SYMDEB вводятся в диалоговом режиме после приглашающего символа -. Группа команд может быть выполнена до начала отладки.

Введенную команду SYMDEB можно отменить нажатием клавиш **CTRL-C** или задержать нажатием клавиш **CTRL-S**. Если отлаживаемая программа обратилась к вводу/выводу, этими клавишами можно отменить или задержать введенную команду **G**. Если программа не обратилась к вводу/выводу, остановить ее выполнение можно только соответствующими клавишами, если были заданы опции **/K** или **/N**.

Некоторые команды имеют параметры, которыми могут быть числа, символы или выражения. Параметры, если их несколько, разделяются запятыми. Между именем команды и первым параметром пробела не требуется, исключая те случаи, где это оговорено особо.

Ниже приведено описание способов кодирования различных типов параметров в командах SYMDEB.

1. Символы могут представлять регистр, абсолютное значение, адрес сегмента или смещение сегмента и состоят из одного или более символов, включая **_**, **?**, **@** и **$**. Первым символом должна быть буква. Все символы должны быть определены в символьном файле. SYMDEB не чувствителен к смене регистра. При совпадении символов с именами регистров последние имеют больший приоритет.

2. Числа представляются в виде:

- **<число>Y** — двоичное
- **<число>O** — 8-ричное
- **<число>Q** — 8-ричное
- **<число>T** — десятичное
- **<число>H** — 16-ричное

Допустимо кодирование ключевых символов **Y**, **O**, **Q**, **T** и **H** на регистре строчных букв. При распознавании чисел максимальный приоритет имеют 16-ричные числа.

3. Адреса представляются в виде:

```
<сегмент>:<смещение>
```

<Сегмент> и **<смещение>** могут быть числами (представимыми как 16-разрядные), именами регистров или символами. Многие команды имеют умалчиваемые имена регистров (**DS** или **CS**).

4. В качестве смещения могут быть указаны границы:

```
<начальный адрес> <конечный адрес>
```

Если **<конечный адрес>** опущен, предполагается значение **<начальный адрес>+128**.

5. В качестве смещения может быть задан счетчик:

```
<начальный адрес> L <число объектов>
```

Размер объекта (байт, слово, инструкция) определяется использующей эту конструкцию командой.

6. Номер линии представляет собой комбинацию десятичных чисел, имен файлов и символов, определяющих уникальную линию исходного текста программы. Номер линии может быть задан в трех различных формах:

```
.+<число> | - <число>
.[<имя файла>:]<число>
.<символ>[+<число> | -<число>]
```

Первая форма определяет смещение (в линиях) относительно текущей линии исходного текста. Вторая форма отображает абсолютный номер линии в файле с указанным именем. Если имя файла опущено, предполагается текущий файл, определяемый текущими значениями регистров **CS** и **IP**. В третьей форме **<символ>** может быть инструкцией или меткой процедуры. Если задано **<число>**, оно является смещением (в линиях) искомой линии относительно линии, идентифицированной указанным символом.

7. Строки являются набором значений в коде ASCII и могут быть заданы в двух форматах:

```
\&'<символы>'
"<символы>"
```

Если в строке присутствуют символы ' или ", их следует кодировать дважды.

8. Выражение представляет собой комбинацию параметров и операторов, которая преобразуется в 8-, 16- или 32-битовое значение. Могут использоваться скобки. Унарные адресные операторы подразумевают регистр **DS** как умалчиваемую адресную базу (сегмент).

Ниже приведены унарные операторы (в порядке убывания приоритетов):

- +
- -
- **NOT** — дополнение операнда до 1
- **SEG** — адрес сегмента операнда
- **OFF** — смещение операнда
- **BY** — младший байт из указанного адреса

- **WO** — младшее слово из указанного адреса
- **DW** — двойное слово из указанного адреса
- **POI** — указатель из указанного адреса (как DW)
- **PORT** — 1 байт из указанного порта
- **WPORT** — слово из указанного порта

Ниже приведены бинарные операторы (в порядке убывания приоритетов):

- *
- / — деление нацело
- **MOD** — modulus
- **:** — переключение сегмента
- +
- -
- **AND** — побитовое И
- **XOR** — побитовое исключающее ИЛИ
- **OR** — побитовое И

Команда ассемблирования

Синтаксис:

```
A[<адрес>]
```

Команда ассемблирует мнемонические инструкции и помещает полученный код по указанному адресу. Если адрес не указан, он определяется содержимым регистров **CS** и **IP**.

При введении команды **A** выдается текущий адрес, и SYMDEB ожидает введения новых инструкций, которые могут кодироваться строчными или заглавными буквами или вперемешку. Инструкции ассемблируются по одной.

Ввод каждой инструкции отмечается нажатием клавиши **ENTER**. SYMDEB ассемблирует последнюю введенную инструкцию и выдает новый адрес. Конец ввода инструкций обозначается нажатием **ENTER** без вводимых символов.

Если введенная инструкция содержит ошибку, об этом выдается сообщение, и текущий адрес не изменяется.

При использовании команды **A** необходимо соблюдать следующие правила:

1. Дальний возврат обеспечивается мнемоникой **RETF**.

2. Инструкции обработки строк должны явно указывать длину строки. Рекомендуется использовать **MOVSB** и **MOVSW**.

3. SYMDEB автоматически ассемблирует короткие, внутренние и дальние скачки и вызовы в зависимости от местоположения целевого адреса. Это правило можно отменить кодированием префиксов **NEAR** (допустимо сокращение **NE**) и **FAR**, например:

```
JMP NEAR 502
JMP FAR 50A
```

4. SYMDEB не может определить, ссылаются ли операнды на слово или на байт памяти. Поэтому тип данных должен быть явно указан префиксами **WORD** (допустимо сокращение **WO**) **PTR** или **BYTE** (**BY**) **PTR**, например:

```
MOV WORD PTR [bp],1
MOV BY PTR [si-1],symbol
```

5. SYMDEB не может определить, ссылается ли операнд на область памяти или это непосредственный операнд. По этой причине ссылка на область памяти должна быть заключена в одинарные квадратные скобки.

6. Для директив **DB** и **DW** ассемблируются байт или слово непосредственно в памяти.

7. SYMDEB поддерживает все формы косвенно-регистровой адресации, например:

```
ADD bx,34[bp+2].[si-1]
pop [bp+di]
push [SI]
```

8. Распознаются все синонимы кодов (для команд перехода). Результатом работы команды реассемблирования **U** являются также синонимы.

9. Не следует ассемблировать и исполнять инструкции процессоров, если система не обеспечена такими сопроцессорами. Например, использование в такой ситуации инструкции **WAIT** может «повесить» систему.

Точки выхода

SYMDEB предоставляет пользователю возможность вставить в тело тестируемой программы набор точек выхода, достижение которых при исполнении кода, вызовет прекращение работы программы, выдачу на дисплей текущего состояния всех флагов и регистров в формате команды **R** и возврат в SYMDEB.

Ниже описаны команды манипуляции с точками выхода.

1. Установка точки выхода. Синтаксис:

```
BP [<число>]<адрес>[<счетчик пропусков>] ["<команды>"]
```

Точка выхода вставляется по указанному адресу. Точки выхода, вставляемые по команде BP, в отличие от точек выхода, создаваемых командой **G**, остаются в теле программы (в памяти, разумеется) до тех пор, пока не будут удалены командой **BC**.

SYMDEB позволяет устанавливать до 10 точек выхода с номерами от 0 до 9. Если специфицировано **<число>**, оно задает номер вставляемой точки, в противном случае подразумевается первая доступная точка.

<Адрес> может определять начало некоторой реальной инструкции.

<Счетчик пропусков> задает число «холостых» выполнений точки выхода, когда ее действие игнорируется, до первого ее срабатывания. Счетчик хранится как 16-битовое число.

<Команды> SYMDEB будут выполняться при каждом срабатывании точки выхода. Друг от друга они должны отделяться символом **;**.

2. Изъятие точек выхода. Синтаксис:

```
BC <список> | *
```

<Список> представляет собой последовательность целых чисел без знака в пределах от 0 до 9. Точки выхода с номерами из этого списка удаляются из программы. Если специфицировано *****, удаляются все точки.

3. Блокировка точек выхода. Синтаксис:

```
BD <список> | *
```

Команда выполняет функции, аналогичные функциям команды **BC**, но точки не удаляются из программы, а временно блокируются до выполнения соответствующей команды **BE**.

4. Разблокировка точек выхода. Синтаксис:

```
BE <список> | *
```

Команда **BE** противоположна по функциям команде **BD**.

5. Выдача списка точек выхода. Синтаксис:

```
BL
```

По этой команде выдается информация о текущем расположении созданных командой **BP** точках выхода, включающая номер точки, статус блокировки, адрес, число оставшихся пропусков, в скобках — исходное значение счетчика пропусков, а также номер линии исходного текста, если такие номера используются.

Статус блокировки может принимать следующие значения: **e** для разблокированной точки, **d** для заблокированной точки или **v** для виртуальной точки. Точка выхода считается виртуальной, если она была вставлена в файл в то время, когда он еще не был загружен.

Комментарии

Синтаксис:

```
* <комментарий>
```

Текст комментария выводится на дисплей.

Команда сравнения

Синтаксис:

```
C <границы> <адрес>
```

По этой команде байты памяти в указанных границах сравниваются с соответствующими им байтами, начиная с указанного адреса.

Если все байты совпадают, SYMDEB опять выдает свой приглашающий символ. В противном случае предварительно выдаются все пары несовпадающих байтов.

Команда «выдать»

Синтаксис:

```
?<выражение>
```

Значение специфицированного выражения вычисляется и выдается на консоль в различных форматах. Выдаваемая информация включает в себя полный адрес, 16-битовое 16-ричное значение, полное 32-битовое 16-ричное значение, десятичное значение в скобках и значение строки в двойных кавычках. Символы строки представляются точками, если их значение меньше 32 (20h) или больше 126 (7Eh).

Выражение может быть комбинацией чисел, символов, адресов и операторов.

Команды дампа

Следующие команды SYMDEB обеспечивают выдачу на консоль дампа оперативной памяти:

1. Дамп памяти по адресу или в границах. Синтаксис:

```
D [<адрес> | <границы>]
```

Формат дампа определяется предыдущей введенной командой дампа. Если же это первая вводимая команда дампа, предполагается формат команды **DB**.

Команда **D** выдает одну или более линий в зависимости от того, **<адрес>** или **<границы>** указаны. Выдается по меньшей мере одно значение. Если специфицированы **<границы>**, выдаются все значения в них. Если операнд опущен, выдается содержимое памяти, начинающейся сразу после области, выданной предыдущей командой дампа. Если раньше дампирования не производилось, используется содержимое регистра **IP**, а если и этот регистр не был определен, используется содержимое регистра **DS**.

2. Дамп памяти по адресу или в границах в коде ASCII. Синтаксис:

```
DA [<адрес> | <границы>]
```

Выдается одна или более линий в зависимости от того, **<адрес>** или **<границы>** указаны. Может быть выдано до 48 символов в линию. Символы, не имеющие аналогов в коде ASCII, то есть, со значением, большим 7Eh или меньшим 20h, обозначаются точками.

Если специфицирован **<адрес>**, выдается содержимое памяти до ближайшего нулевого байта или 128 байтов. Если параметр опущен, выдаются значения всех байтов, начиная с непосредственно следующего за выданным предыдущей командой дампа, до ближайшего нулевого или 128 байтов. Если при задании границ использовано **L**, выдается указанное число байтов.

3. Дамп памяти по байтам. Синтаксис:

```
DB [<адрес> | <границы>]
```

Выдается комбинированный (16-ричный и символьный в коде ASCII) дамп памяти, начиная с указанного адреса или в указанных границах. Если указан **<адрес>** выдаются значения 128 байтов.

4. Дамп памяти по словам. Синтаксис:

```
DW [<адрес> | <границы>]
```

Аналогично **DB**, но при указании адреса выдается содержимое 64 слов памяти.

5. Дамп памяти по двойным словам. Синтаксис:

```
DD [<адрес> | <границы>]
```

Аналогично **DB**, но при указании адреса выдается содержимое 32 двойных слов памяти.

6. Дамп коротких десятичных чисел. Синтаксис:

```
DS [<адрес> | <границы>]
```

ыдается комбинированный (16-ричный и в форме коротких, 4-байтных десятичных чисел с плавающей точкой) дамп памяти, начиная с указанного адреса, или в указанных границах.

Выдается одна или более линий в зависимости от того, **<адрес>** или **<границы>** указаны.

Выдается по меньшей мере одно число. Если специфицированы границы, выдаются все числа в их пределах.

7. Дамп длинных десятичных чисел. Синтаксис:

```
DL [<адрес> | <границы>]
```

Команда аналогична **DS**, но оперирует с длинными (8 байтов) десятичными числами.

8. Дамп 10-байтных десятичных чисел. Синтаксис:

```
DT [<адрес> | <границы>]
```

Команда аналогична **DS**, но оперирует с 10-байтными десятичными числами.

Команды ввода с клавиатуры

При помощи команд ввода с клавиатуры данные могут быть введены непосредственно в память.

1. Ввод значений. Синтаксис:

```
E <адрес> [<список>]
```

Одно или более значений вводятся в память, начиная с указанного адреса. Размер значений устанавливается предыдущей командой ввода с клавиатуры, а если ее не было, предполагается **EB**.

Если элемент списка задан некорректно, список игнорируется.

Если список опущен, SYMDEB запрашивает значения в диалоговом режиме с указанием текущего адреса. Конец списка в этом случае обозначается нажатием клавиши **ENTER** без указания значения.

2. Ввод байтов. Синтаксис:

```
EB <адрес> [<список>]
```

Одно или более чисел, занимающих 1 байт, вводятся в память, начиная с указанного адреса.

Если список опущен, SYMDEB запрашивает значения в диалоговом режиме с указанием текущего адреса. При этом имеется возможность заменить или пропустить значение, вернуться к предыдущему значению, выйти из команды.

Для замены числа следует записать новое значение после текущего.

Для пропуска числа необходимо нажать **SPACE**.

Для возврата к предыдущему числу следует ввести **-**. Для выхода из команды служит клавиша **ENTER**.

3. Ввод байтов. Синтаксис:

```
EA <адрес> [<список>]
```

Аналогично **EB**.

4. Ввод слов. Синтаксис:

```
EW <адрес> [<значение>]
```

Указанное значение в формате слова вводится в память по указанному адресу. После ввода SYMDEB запрашивает значение следующих 4 байтов. Конец ввода обозначается нажатием клавиши **ENTER** без указания значения.

5. Ввод двойных слов. Синтаксис:

```
ED <адрес> [<значение>]
```

Указанное значение в формате двойного слова вводится в память по указанному адресу. Двойное слово кодируется как пара слов, разделенных символом **:**. После ввода SYMDEB запрашивает значение следующих 8 байтов. Конец ввода обозначается нажатием клавиши **ENTER** без указания значения.

6. Ввод коротких десятичных чисел. Синтаксис:

```
ES <адрес> [<значение>]
```

Указанное значение в формате короткого (4 байта) десятичного числа с плавающей точкой вводится в память по указанному адресу. После ввода SYMDEB запрашивает значение следующих 4 байтов. Конец ввода обозначается нажатием клавиши **ENTER** без указания значения.

7. Ввод длинных десятичных чисел. Синтаксис:

```
EL <адрес> [<значение>]
```

Команда аналогична **ES**, но оперирует с длинными (8 байтов) десятичными числами, которые представляются парой чисел, разделенных символом **:**.

8. Ввод 10-байтных десятичных чисел. Синтаксис:

```
ET <адрес> [<значение>]
```

Команда аналогична **EL**, но оперирует с 10-байтными десятичными числами.

Просмотр символьного плана

Синтаксис:

```
X [*]
X? [<имя плана>!][<имя сегмента>:][<символ>]
```

SYMDEB создает символьный план для каждого символьного файла, специфицированного в командной строке запуска SYMDEB.

Форма команды **X** обеспечивает выдачу имен и загрузочных адресов сегментов и символов текущего символьного плана. Если специфицировано *****, выдаются имена и адреса всех загруженных в данный момент символьных планов.

Форма команды **X?** обеспечивает выдачу одного или более символов в символьном плане. Если указано имя символьного плана, выдается информация об этом плане (**<имя символьного плана>** должно представлять имя соответствующего файла без расширения). Если задано имя сегмента, выдаются имя и загрузочный адрес этого сегмента. Если специфицирован символ, выдаются адрес сегмента и смещение этого символа. Для того, чтобы получить информацию о нескольких символах или сегментах, следует задавать частичные имена с использованием символа *****.

Команда «наполнения»

Синтаксис:

```
F <границы> <список>
```

Память в указанных границах «наполняется» значениями из указанного списка. Если границы определяют больше памяти, чем может занять список, список начинает обрабатываться сначала. Если же длиннее список, его не помещающийся в границы остаток игнорируется.

Команда Go

Синтаксис:

```
G [=<адрес начала программы>][<адреса точек выхода>]
```

По команде **G** управление передается программе по указанному адресу начала. Выполнение продолжается до конца программы или до достижения точек выхода, если они указаны. Программа останавливается также на точках выхода, внесенных командой **BP**.

Если адрес начала программы опущен, управление передается по адресу, определяемому содержимым регистров **CS** и **IP**.

Для передачи управления программе используется инструкция **IRET**. При этом SYMDEB соответствующим образом устанавливает указатель стека пользователя и заносит в этот стек флаги и регистры **CS** и **IP**. Следует иметь в виду, что если стек пользователя не содержит хотя бы 6 байтов доступной памяти, выполнение команды G может «повесить» систему.

Все значения без предшествующего **=** (знак равенства) рассматриваются как адреса точек выхода. Допустимо задание до 10 точек выхода. Их адреса должны указывать на начало инструкции.

Для создания точки выхода SYMDEB по каждому специфицированному адресу помещает инструкцию **INT** с кодом прерывания 204 (0CCh). При выходе из программы в одной из этих точек все внесенные инструкции **INT** удаляются из программы. Однако, если выполнение продолжается до конца программы или будет прекращено каким-либо другим образом, SYMDEB не удаляет внесенные им инструкции. В этом случае до следующего запуска нужно перезагрузить программу при помощи команд **N** и **L**.

Когда выполнение программы достигает точки выхода, SYMDEB выдает на дисплей текущее содержимое всех регистров и флагов в формате команды **R**, а также следующую инструкцию. Выполнение программы прекращается.

Если выполнение достигло логического конца программы, SYMDEB выдает сообщение **Program terminated normally**, и на экран выдается текущее содержимое регистров и флагов.

Справка о командах SYMDEB

Синтаксис:

```
?
```

На дисплей выдается список команд SYMDEB.

HEX-команда

Синтаксис:

```
H <значение 1> <значение 2>
```

Выполняются операции **<значение 1>+<значение 2>** и **<значение 1>-<значение 2>**, и результаты выдаются на экран в виде 16-ричных чисел.

Команда ввода из порта

Синтаксис:

```
I <номер порта>
```

Из порта с указанным номером (представимым 16-битным числом) считывается и выдается на экран 1 байт данных.

Команда загрузки файла

Синтаксис:

```
L [<адрес> [<драйв> <запись> <счетчик>]]
```

Команда **L** обеспечивает считывание файла с диска и запись его в память.

Если опущены все параметры команды, загружается файл с именем, ранее определенным командой **N** или заданным аргументом при запуске SYMDEB. Если имя не было определено, SYMDEB считывает его из области памяти по адресу **DS:5C**. Эта область представляет собой управляющий блок, содержащий имя определенного по команде **N** файла или аргумент запуска.

Пара регистров **BX:CX** должна содержать число загружаемых байтов.

В памяти файл размещается, начиная с указанного адреса или, если он опущен, с адреса **CS:100**.

Если специфицированы все 4 параметра команды **L**, в память по указанному адресу загружается определяемое счетчиком количество логических записей, расположенных на указанном драйве. **<Запись>** определяет первую считываемую запись и может представляться 16-ричным числом, содержащим от 1 до 4 16-ричных цифр. **<Счетчик>** представляется аналогично. **<Драйв>** может быть задан числом 0, 1, 2 или 3, определяющим логический адрес драйва A, B, C или D соответственно.

Если имя файла имеет расширение **EXE**, его длина и адрес загрузки берется из заголовка файла, а параметры **<адрес>** и **<счетчик>** команды **L**, если они указаны, игнорируются.

Если имя файла имеет расширение **HEX**, адрес загрузки определяется суммой указанного в заголовке файла адреса и параметра **<адрес>** команды **L** или, если этот параметр опущен, только адресом загрузки из заголовка.

Команда перемещения

Синтаксис:

```
M <границы> <адрес>
```

Блок памяти в указанных границах перемещается по заданному адресу.

Перемещение произойдет нормально, даже если исходный и принимающий блоки перекрываются. Принимающий блок всегда является точной копией исходного. При перекрытии содержимое исходного блока может измениться.

Для предотвращения потери данных команда **M** побайтно копирует данные, начиная со старших адресов исходного блока, если адрес его начала меньше адреса начала принимающего блока. В противном случае копирование производится, начиная с младших адресов исходного блока.

Команда установки имени

Синтаксис:

```
N [<имя файла>] [<аргументы>]
```

Устанавливается имя файла для последующего выполнения команд **L** и **W** или аргументы для последующего исполнения программы.

Аргументы, если они заданы, копируются, включая пробелы, в область памяти по адресу **DS:81**. Длина поля параметров запоминается в байте памяти по адресу **DS:80**.

Если первые два аргумента являются именами файлов, по адресам **DS:5C** и **DS:6C** создаются блоки управления файлами (**FCB**), куда по соответствующим форматам и копируются имена.

Команда **N** трактует **<имя файла>** как тоже аргумент, записывая его в память по адресу **DS:81** и создавая **FCB** по адресу **DS:5C**. Поэтому необходимо помнить, что установка нового значения имени файла изменяет предыдущие аргументы программы.

Команда открытия символьного плана

Синтаксис:

```
XO [<имя плана>!][<имя сегмента>]
```

Команда **XO** устанавливает активным символьный план и/или сегмент.

Имя символьного плана, если оно указано, должно быть именем (без расширения) специфицированного при запуске SYMDEB символьного файла.

Имя сегмента, если оно указано, должно быть именем сегмента в специфицированном символьном плане. Все сегменты в открытом плане доступны, но открывается первый сегмент.

Команда вывода в порт

Синтаксис:

```
O <номер порта> <1-байтовое 16-ричное значение>
```

Указанное значение направляется в порт с данным номером. Номер порта должен быть представим 16-битовым значением.

Р-трассировка

Синтаксис:

```
P [=<адрес>] [<счетчик>]
```

Команда **P** выполняет задаваемую адресом инструкцию и затем выдает на дисплей в формате команды **R** текущее содержимое всех регистров и флагов.

Если **<адрес>** опущен, выполняется инструкция по адресу, определяемому регистрами **CS** и **IP**.

Счетчик, если он присутствует, задает число инструкций, которые будут выполнены до останова. Выдача регистров и флагов при этом будет осуществляться после выполнения каждой инструкции до начала выполнения следующей.

Р-трассировка подобна Т-трассировке с той лишь разницей, что Р-трассировка прекращается после вызова процедуры или прерывания в то время, как Т-трассировка отслеживает и такие ситуации.

Формат и объем выдаваемой информации существенно зависит от режима, установленного командой **S**.

Команда выхода из SYMDEB

Синтаксис:

```
Q
```

По этой команде SYMDEB заканчивает свою работу.

Команды переназначения ввода/вывода

Синтаксис:

```
<<имя устройства>
> <имя устройства>
=<имя устройства>
<имя устройства>
<имя устройства>
~<имя устройства>
```

Команды переназначения блокируют последующие операции ввода/вывода и направляют их на работу с указанным устройством. Это может понадобиться, например, когда отлаживаемая программа выводит графическую информацию на консоль, используемую и SYMDEB.

Команда > переключает весь последующий вывод SYMDEB, а команда < — весь последующий ввод SYMDEB на указанное устройство. Команда = включает в себя обе эти функции.

Команда переключает весь последующий ввод отлаживаемой программы, а команда - весь последующий вывод отлаживаемой программы на указанное устройство. Команда ~ включает в себя обе эти функции.

В качестве имени устройства может быть указано принятое в DOS имя устройства или имя файла.

Если в качестве имени устройства задано COM1 или COM2, это накладывает дополнительные ограничения на конфигурацию системы. В частности, в этом случае становятся недоступными и игнорируются комбинации клавиш **CTRL-C** и **CTRL-S**.

Выдача регистров

Синтаксис:

```
R [<имя регистра>[[=]<значение>]]
```

Команда **R** выдает на дисплей содержимое регистров процессора и позволяет загружать их требуемыми значениями.

Если имя регистра опущено, выдаются все регистры и флаги.

Кроме того, выдается инструкция, на которую указывают регистры **CS** и **IP**, и может быть сделана попытка выполнить ее. При этом SYMDEB вычисляет операнды инструкции. Если инструкцией является вызов DOS, будет показан номер функции. Если регистры **CS** и **IP** установлены не на инструкцию, а на позицию в памяти, будет выдан адресующий ее символ.

Если указано имя регистра, SYMDEB выдает его содержимое и запрашивает новое значение (ответ заключается в наборе требуемых символов и нажатии клавиши **ENTER**). Если заданы и имя регистра, и значение, производится загрузка регистра.

Могут быть указаны следующие имена регистров: **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **CS**, **DS**, **SS**, **ES**, **IP**, **PC** (**IP**) и **F** (флаги).

Установка значений флагов осуществляется следующим образом. При спецификации **F** в команде **R** состояние каждого флага выдается в виде двухсимвольного кода. Список значений заканчивается символом **-**, после которого можно в этом же коде в любой последовательности ввести новые значения выбранных флагов. Значения остальных флагов не изменятся. Каждый флаг в одном списке может быть специфицирован только один раз. Конец списка обозначается нажатием клавиши **ENTER**.

	Коды значений флагов	
Флаг	1	0
Переполнение	OV	NV
Направление	DN (уменьшение)	UP (увеличение)
Прерывания	EI (запрет)	DI (разрешение)
Знак	NG (минус)	PL (плюс)
Нуль	ZR	NZ
Вспомогательный перенос	AC	NA
Паритет	PE (четность)	PO (нечетность)
Перенос	CY	NC

Флаги модифицируются последовательно по одному. Поэтому при ошибочном кодировании возникнет ситуация, когда часть флагов (до ошибочного символа) изменила свои значения, а часть, начиная с ошибочного символа, — нет.

Формат и объем выдаваемой информации существенно зависит от режима, установленного командой **S**.

Смена экрана

Синтаксис:

```
\
```

Эта команда позволяет переключиться с экрана отладки на экран отлаживаемой программы, что бывает нужно, когда программа, например, выводит на экран графическую информацию.

Возврат на экран SYMDEB осуществляется нажатием любой клавиши.

При работе на IBM-машине для использования команды смены экрана необходимо задание опции SYMDEB **/S**. При работе на машине, совместимой с IBM, необходимо также задать опцию **/I**.

Команда поиска

Синтаксис:

```
S <границы> <список>
```

В области памяти с указанными границами осуществляется поиск заданных в списке 1-байтных значений с индикацией на экране результатов поиска. Элементы списка должны разделяться запятыми.

Команда установки режима индикации

Синтаксис:

```
S-|+|&
```

Команда **S** устанавливает режим выдачи на дисплей, которому SYMEB будет в дальнейшем придерживаться.

По команде **S+** SYMDEB будет выдавать на экран информацию в терминах линий исходного текста программы. Команда **S-** устанавливает режим выдачи реассемблированного кода программы. Команда **S&** обеспечивает обе эти функции. Первоначальным умалчиваемым режимом является **S&**.

Команда **S** имеет смысл лишь при работе с программами на языке высокого уровня. Для ассемблерных программ автоматически устанавливается режим **S-**.

Если символьный план не открыт или не содержит информацию о линиях исходной программы, SYMDEB игнорирует все последующие запросы на выдачу исходных линий. В режиме **S&** SYMDEB выдает исходную линию, содержащую инструкцию, на которую указывает пара регистров **CS** и **IP**.

Команда **S** оказывает влияние на работу последующих команд реассемблирования **U**.

Команда **S** влияет также на работу команд **R**, **T** и **P**. В режиме **S+** эти команды выдают за раз одну исходную линию, которая может соответствовать нескольким реассемблированным инструкциям. В режиме **S-** выдается только реассемблированный код. В режиме **S&** выдаются реассемблированный код и номера линий.

Исходные линии выдаются в виде:

```
<номер линии>: <исходный текст>
```

Исходные линии при выдаче предшествуют реассемблированным инструкциям.

Всякий раз, когда SYMDEB первый раз обращается к исходному тексту, он ищет в текущем подоглавлении файл с тем же базовым именем, что и соответствующий символьный файл. При неудачном поиске SYMDEB выдает на консоль запрос об имени исходного файла. Если в ответ на него нажать клавишу **ENTER**, не указав имени, SYMDEB подавит выдачу исходных линий, вместо которых тогда будут выдаваться имя плана и номер линии. Эту процедуру нужно производить при работе с программами, сформированными ранними версиями (до 3.31) компиляторов FORTRAN и PASCAL.

Временный выход в DOS

Синтаксис:

```
! [<команда DOS>]
```

Команда **!** позволяет выполнить **COMMAND.COM** и команды MS-DOS вне SYMDEB. **COMMAND.COM** выполняется без аргументов с сохранением контекста отладки. Для возврата в SYMDEB служит команда MS-DOS **EXIT**.

Если в команде **!** задана команда DOS, производится выполнение этой команды и автоматический возврат в SYMDEB.

Использование команды **!** требует дополнительной памяти. Для более экономного использования памяти рекомендуется предусматривать в ассемблерных программах вызов DOS с функцией 4Ah, который освободит неиспользуемую память. Тот же эффект дает опция LINK **/CPARMAXALLOC**. Программы на языке С, обработанные компилятором MICROSOFT С версии 3.0 и выше, освобождают ненужную память автоматически, если была выполнена процедура **_main**. SYMDEB также освобождает неиспользуемую им память.

В команде **!** не может быть использован ограничитель **;**, так как весь текст после **!** передается в **COMMAND.COM** и будет интерпретироваться как командная строка DOS.

Для размещения копии **COMMAND.COM** SYMDEB использует переменную **COMSPEC** команды DOS **SET**.

Команда выдачи кода исходной линии

Синтаксис:

```
.
```

По этой команде выдается код исходной линии невзирая на режим, установленный командой **S**.

Команда не может быть использована при работе с ассемблерными программами.

Команда трассировки стека

Синтаксис:

```
K [<число>]
```

Эта команда позволяет увидеть текущий кадр стека. Первая строка трассы содержит имя текущей процедуры, ее аргументы, имя файла и номер линии, вызвавшей процедуру. Следующая строка описывает вызвавшую процедуру. Если число аргументов процедуры переменно или неизвестно, SYMDEB использует специфицированное в команде **K** число, которое задает число слов параметров.

Команда **K** работает только при соблюдении стандартных соглашений о связях, в противном случае она игнорируется.

Команда установки символа

Синтаксис:

```
Z <символ> <значение>
```

В результате выполнения команды **Z** указанный символ будет связан со специфицированным значением адресом.

T-трассировка

Синтаксис:

```
T [=<адрес начала>] [<счетчик>]
```

Команда **T** выполняет задаваемую адресом инструкцию и затем выдает на дисплей в формате команды **R** текущее содержимое всех регистров и флагов.

Если **<адрес>** опущен, выполняется инструкция по адресу, определяемому регистрами **CS** и **IP**.

Счетчик, если он присутствует, задает число инструкций, которые будут выполнены до останова. Выдача регистров и флагов при этом будет осуществляться после выполнения каждой инструкции до начала выполнения следующей.

В отличие от P-трассировки T-трассировка не прекращается при вызове процедуры или прерывании. Исключение составляет лишь прерывание с номером 21h (функция DOS).

Трассировка будет идти лучше, если не использовались средства оптимизации данного языка.

С помощью команды **T** можно трассировать инструкцию в ROM (read-only memory).

Формат и объем выдаваемой информации существенно зависит от режима, установленного командой **S**.

Команда реассемблирования

Синтаксис:

```
U [<границы>]
```

По команде **U** на дисплей выдаются реассемблированные инструкции и/или предложения исходной программы. Формат вывода зависит от режима, установленного командой **S**, и от языка, на котором была написана программа. Если отлаживаемая программа была обработана MASM или несовместимым с SYMDEB компилятором, принудительно устанавливается режим **S-**. В режимах **S+** и **S&** при работе с программами, выработанными совместимыми с SYMDEB компиляторами, по команде **U** выдаются линии исходного текста и реассемблированные инструкции (одна исходная линия для каждой соответствующей группы предложений Ассемблера). Исходные линии считываются из исходного файла. Инструкции для реассемблирования берутся из блока памяти в указанных в команде границах.

В отличие от команд **T** и **P**, для команды **U** режимы **S+** и **S&** совпадают. Для обоих режимов (исходного и смешанного) SYMDEB требует, чтобы вместе с программой был загружен и символьный план, содержащий информацию о номерах строк исходной программы. При отсутствии этой информации исходные линии не выдаются.

Если **<границы>** опущены, обрабатываются первые 8 линий кода по текущему адресу реассемблирования. Текущим адресом реассемблирования является адрес первого байта (линии) после последнего байта

(линии), выданного предыдущей командой **U**. Защищенные инструкции процессора 80286 не могут быть реассемблированы.

Команда выдачи исходной линии

Синтаксис:

```
V <адрес>
```

Команда **V** выдает исходные линии программы, соответствующие инструкциям, расположенным, начиная с указанного адреса.

При этом загруженный символьный план должен содержать информацию о линиях исходной программы.

Исходные линии выдаются независимо от режима, установленного командой **S**.

Команда записи на диск

Синтаксис:

```
W [<адрес> [<драйв> <запись> <счетчик>]]
```

Команда **W** обеспечивает запись участка памяти в определенное место на диске.

Если опущены все параметры команды, запись производится в файл с именем, ранее определенным командой **N**.

Пара регистров **BX:CX** должна содержать число записываемых байтов.

В памяти записываемые данные должны размещаться, начиная с указанного адреса или, если он опущен, с адреса **CS:100**.

Если специфицированы все 4 параметра команды **W**, на указанный драйв записывается определяемое счетчиком количество логических записей, содержимое которых расположено по указанному адресу. **<Запись>** определяет первую записываемую на диск запись и может представляться 16-ричным числом, содержащим от 1 до 4 16-ричных цифр. **<Счетчик>** представляется аналогично. **<Драйв>** может быть задан числом 0, 1, 2 или 3, определяющим логический адрес драйва A, B, C или D соответственно.

Не рекомендуется записывать данные по абсолютному адресу сектора диска, даже если имеется уверенность, что сектор свободен. Запись на зарезервированный или занятый сектор может испортить содержимое файла или даже диска.

Если имя отлаживаемого файла имеет расширение **COM** или **BIN**, можно при помощи SYMDEB внести изменения в программу и затем записать ее обратно в файл. При загрузке файла его длина, начальный адрес и имя будут установлены правильно с точки зрения последующей записи. Однако, если в процессе отладки использовались команды **G**, **P** или **T** или изменялось содержимое пары регистров **BX:CX**, утраченные значения следует восстановить до записи.

Команду **W** нельзя использовать для записи в **EXE-** или **HEX-**файл. Для модификации таких файлов может служить следующая процедура:

1. Запустить SYMDEB с исполнительным файлом и запомнить несколько первых инструкций программы.

2. Выйти из SYMDEB и переименовать исполнительный файл так, чтобы расширение его имени отличалось от **EXE** и **HEX**.

3. Запустить SYMDEB с переименованным исполнительным файлом. При этом заголовок файла будет рассматриваться как часть кода программы (очевидно, что в этом случае нет смысла в загрузке символьных файлов, так как все символы в них будут иметь некорректные адреса).

4. Использовать команду поиска **S** для нахождения действительного начала программы по запомненным инструкциям. Для этого может понадобиться несколько попыток, так как начальный адрес может меняться в зависимости от порядка сегментов и других факторов.

5. Имея адрес начала программы, отыскать инструкции, в которые нужно внести изменения, и проделать эти изменения.

6. Установить параметры команды **W** и записать весь файл, включая его заголовок, на диск. Длина заголовка должна входить в общую длину записываемого файла в паре регистров **BX:CX**.

7. Выйти из SYMDEB и произвести обратное переименование исполнительного файла.



CREF: утилита перекрестных ссылок

Утилита перекрестных ссылок CREF предназначена для создания листинга перекрестных ссылок всех символов в ассемблерной программе. Для каждого символа указываются номера строк исходной программы, содержащих ссылки на этот символ.

Листинг перекрестных ссылок вместе с создаваемой Ассемблером таблицей символов упрощает отладку программы.

Листинг перекрестных ссылок создается на основе формируемого Ассемблером CRF-файла.

Запуск CREF

Утилита CREF может быть запущена двумя способами:

- С использованием подсказок.
- При помощи командной строки.

Для запуска CREF с использованием подсказок необходимо ввести командную строку, содержащую только имя файла CREF и спецификацию его пути поиска, если это необходимо. CREF перейдет в диалоговый режим и серией подсказок запросит у пользователя информацию о следующих файлах (ответ заключается в наборе требуемых символов и нажатии клавиши **ENTER**):

1. Имя файла перекрестных ссылок, сформированного Ассемблером. Если при ответе не указано расширение, предполагается **CRF**.

2. Имя файла создаваемого файла листинга перекрестных ссылок. Если при ответе не указано расширение, предполагается **REF**.

Для запуска CREF при помощи командной строки необходимо ввести командную строку следующего вида:

```
CREF <имя файла перекрестных ссылок>
[,<имя файла листинга перекрестных ссылок>][;]
```

Умалчиваемые расширения имен файлов совпадают со случаем запуска CREF с использованием подсказок.

Если после имени файла перекрестных ссылок специфицировано **;**, базовое имя файла листинга по умолчанию устанавливается таким же, как и у файла перекрестных ссылок.

Имена файлов при обоих способах запуска могут содержать спецификации путей поиска в подоглавлениях. Если указание пути отсутствует, файл ищется или создается в текущем подоглавлении.



LIB: утилита обслуживания библиотек

Библиотека представляет собой набор объектных модулей, объединенных в одном файле.

Библиотека может быть использована программой LINK для согласования внешних ссылок.

LIB создает для библиотеки таблицу содержимого, где располагаются имена объектных модулей. LINK выбирает из библиотеки только те модули, на которые имеются ссылки в обрабатываемой им программе.

LIB выполняет с библиотечными файлами следующие операции:

- Создание новой библиотеки.
- Проверка содержимого существующей библиотеки.
- Печать листинга библиотечных ссылок.
- Сопровождение библиотек.

Запуск LIB

Запуск LIB может быть осуществлен тремя способами:

- С использованием подсказок.
- При помощи командной строки DOS.
- С использованием файла ответа.

Для запуска LIB с использованием подсказок необходимо ввести командную строку, содержащую только имя программы LIB со спецификацией подоглавления, если она требуется. LIB перейдет в диалоговый режим и серией подсказок запросит у пользователя информацию о

следующих объектах (ответ заключается в наборе требуемых символов и нажатии клавиши **ENTER**):

1. Имя библиотеки, с которой будет производиться работа. Если при ответе не указано расширение, предполагается LIB. Если библиотеки с введенным именем не существует, LIB выдаст запрос:

Library file does not exist. Create?

Ответ y обеспечит создание библиотеки, **n** — возврат в DOS. В этом ответе может быть задана опция **/PAGESIZE**.

2. Операции с библиотекой. Ответом является набор команд LIB. Если команды LIB не помещаются на строке, в ее последней позиции следует поставить признак продолжения — символ **&** и нажать **ENTER**, после чего можно будет продолжать ввод команд.

3. Имя файла листинга. Если не было задано никаких модификаций библиотеки, LIB создает файл листинга и возвращает управление в DOS.

4. Имя выходной библиотеки. Этот запрос появляется в том случае, когда была специфицирована хотя бы одна операция модификации библиотеки.

Если при ответе не указано расширение, предполагается LIB. Библиотека с указанным именем будет создана как копия рабочей библиотеки и все операции будут производиться именно с ней.

Если нажать **ENTER**, не введя имени, операции будут производиться с рабочей библиотекой. В этом случае для старой библиотеки будет создана копия с расширением **BAK**.

Если в каком-либо ответе после первого встречается символ **;**, LIB устанавливает оставшуюся входную информацию по умолчанию.

В любом ответе могут быть заданы ответы на последующие запросы в формате командной строки для запуска LIB.

Для запуска LIB посредством командной строки, необходимо ввести командую строку следующего вида:

```
LIB <имя старой библиотеки> [/PAGESIZE:<число>][<команды>]
[,<имя файла листинга>][,<имя новой библиотеки>]]][;]
```

Назначение библиотек и правила умолчания аналогичны случаю запуска LIB с использованием подсказок.

Символ **;** обозначает конец строки и должен кодироваться последним, если он есть. Оставшаяся неопределенной информация устанавливается по умолчанию.

Спецификации исходной информации LIB могут быть заранее занесены в специальный файл ответа. Имя этого файла с предшествующим символом **@** и указанием пути поиска, если он нужен, может быть помещено в любом месте ответа на подсказку или командной строки и трактуется, как если бы содержимое файла ответа было непосредственно вставлено в это место. Следует, однако, помнить, что комбинация символов **CARRIAGE-RETURN/LINE-FEED** в файле ответа интерпретируется как **ENTER** в подсказке или запятая в командном файле.

Общий вид файла ответа:

```
<имя библиотеки>[/PAGESIZE:<число>]
[<команды>]
[<имя файла листинга>]
[<имя выходной библиотеки>]
```

Каждая группа данных должна задаваться на отдельной строке. Если группа не помещается на одной строке, в последней позиции строки должен стоять признак продолжения — символ **&**.

В файле ответа могут быть опущены компоненты, уже определенные ответами на подсказки или командной строкой.

При обнаружении в файле ответа символа **;** остаток файла игнорируется, и оставшаяся неопределенной информация устанавливается по умолчанию.

При использовании файла ответа его содержимое выдается на консоль в форме подсказок. Если определена не вся информация LIB переходит в диалоговый режим.

Если файл ответа не содержит комбинации символов

CARRIAGE-RETURN/LINE-FEED или символа **;**, LIB выдает на консоль последнюю строку файла и ожидает нажатия **ENTER**.

Единственная опция LIB, задаваемая при имени рабочей библиотеки, определяет размер страницы библиотеки и имеет вид:

```
/PAGEZIZE:<число> или /P:<число>
```

Указанное число задает размер страницы библиотеки в байтах и должно быть целым четным числом в пределах от 2 до 32768. По умолчанию принимается 128 для новой библиотеки или размер страницы существующей библиотеки.

Размер страницы влияет на выравнивание хранимых в библиотеке модулей. Модули всегда располагаются с начала страницы, считая от начала файла.

Из-за индексной технологии поиска и хранения, реализуемой LIB, библиотека с большим размером страницы может содержать большее число модулей. Однако, при этом возможен значительный перерасход памяти на диске. Рекомендуется создавать библиотеки с малым размером страницы.

Имя каждого файла может сопровождаться информацией о подоглавлении, содержащем этот файл, иначе поиск исходного файла или создание выходного файла будет осуществляться в текущем подоглавлении.

Работа LIB может быть в любой момент прекращена нажатием клавиш **CTRL-C**.

Функции и команды LIB

Среди операций с библиотеками, выполняемых LIB, следует различать функции LIB и команды LIB.

Функции LIB не приводят ни к каким модификациям существующих данных.

Команды LIB предназначены для модификации библиотек.

Выполнение команд всегда влечет за собой создание резервной копии исходной библиотеки, хранящей состояние библиотеки до начала коррекций. Именно команды, а не функции LIB вводятся в ответе на подсказку, в командной строке и в файле ответа.

Создание новой библиотеки

Новая библиотека создается при запуске LIB в случае указания имени несуществующей библиотеки и утвердительном ответе на подтверждающий запрос при использовании подсказок.

Если используются команды модификации библиотеки, в новую библиотеку перед началом модификаций копируется содержимое исходной библиотеки, которая в этом случае считается резервной копией.

Команды LIB выполняются с новой библиотекой.

Проверка содержимого библиотеки

Проверка содержимого библиотеки выполняется при задании во всех режимах запуска только имени библиотеки и символа **;**. Она заключается в проверке корректности всех входов библиотеки, что может производиться, например, после перемещения библиотеки.

LIB автоматически осуществляет проверку содержимого каждого добавляемого в библиотеку модуля.

Листинг перекрестных ссылок библиотеки

Файл листинга перекрестных ссылок создается при отсутствии команд LIB и спецификации имени файла в соответствующих подсказке, позиции командной строки или строке файла ответа.

Листинг перекрестных ссылок содержит 2 списка:

- Список всех общих символов в библиотеке с указанием имен содержащих их модулей.
- Список модулей библиотеки с указанием содержащихся в них общих символов.

Команды LIB

Команды LIB служат для сопровождения библиотек и обеспечивают добавление, удаление, замену модулей в библиотеке, а также копирование и перемещение модулей в новые библиотеки.

1. Добавление модуля в библиотеку.

Синтаксис:

```
+<имя объектного файла>
```

Модуль, находящийся в указанном объектном файле, имя которого, если нужно дополнено описанием пути поиска в подоглавлениях, добавляется в текущую библиотеку.

Если не указано расширение имени объектного файла, предполагается **OBJ**.

Модуль помещается в библиотеку под именем, совпадающим с базовым именем объектного файла.

Между знаком **+** и именем файла не должно быть пробелов.

2. Удаление модуля из библиотеки.

Синтаксис:

```
-<имя модуля>
```

Модуль с указанным именем удаляется из текущей библиотеки.

Следует иметь в виду, что команды удаления всегда отрабатываются до команд добавления независимо от их порядка в командной строке. Такой порядок спасает LIB от попыток замены существующей версии модуля на новую версию.

Имена модулей могут задаваться как на регистре строчных букв, так и на регистре заглавных букв.

3. Замена модуля библиотеки.

Синтаксис:

```
-+<имя модуля>
```

Модуль с указанным именем замещается модулем из объектного файла, имеющим то же базовое имя, что и указанное имя, и расширение **OBJ**. LIB сначала удаляет модуль, а затем ищет файл.

Имена модулей могут задаваться как на регистре строчных букв, так и на регистре заглавных букв.

4. Копирование модуля.

Синтаксис:

```
*<имя модуля>
```

Модуль с указанным именем копируется из библиотеки в объектный файл, создаваемый в текущем подоглавлении и имеющий базовое имя, совпадающее с именем модуля, и расширение **OBJ**.

Имена модулей могут задаваться как на регистре строчных букв, так и на регистре заглавных букв.

5. Перемещение модуля.

Синтаксис:

```
-*<имя модуля>
```

Работа этой команды аналогична копированию с той лишь разницей, что после копирования модуль удаляется из библиотеки.

6. Объединение библиотек.

Синтаксис:

```
+<имя библиотеки>
```

Содержимое указанной библиотеки добавляется в текущую библиотеку. Следует помнить, что в этом случае расширение имени опускать нельзя, так как тогда указанное имя будет интерпретироваться как имя объектного файла.

Модули помещаются в конец текущей библиотеки. Исходная библиотека не изменяется.

Этой командой в библиотеки MS-DOS могут быть добавлены библиотеки **XENIX** или **INTEL**.



MAKE: утилита сопровождения программ

Использование утилиты сопровождения программ MAKE позволяет автоматизировать процесс разработки и эксплуатации программ на Ассемблере и языках высокого уровня.

После того, как были внесены изменения в исходный файл, при помощи MAKE могут быть выполнены действия, необходимые по отображению этих изменений в выходных файлах.

В отличие от других программ пакетной обработки, MAKE сравнивает даты последних модификаций выходных (целевых) файлов с датами последних модификаций исходных (требуемых) файлов. MAKE выполняет поставленную задачу, только если целевой файл старше. Это может сэкономить много времени, например, при разработке программ, содержащихся во многих исходных файлах или требующих нескольких шагов компиляции.

Запуск и особенности работы MAKE

Перед запуском MAKE должен быть создан специальный файл описаний MAKE, содержимое которого задает поставленную задачу и определяет требуемые для ее выполнения файлы.

Файл описаний состоит из одного или нескольких описаний цели/источника. Каждое описание задается в виде:

```
<имя исходного файла> : <имена требуемых файлов>
<команда>
...
```

Предполагается, что указанный исходный файл может быть преобразован, для чего могут понадобиться файлы со специфицированными после **:** именами.

Имена файлов при необходимости могут быть снабжены спецификациями путей поиска в подоглавлениях.

<Команда> рассматривается как имя исполнительного файла или команда MS-DOS.

Может быть задано любое число требуемых файлов, но только один исходный (целевой). Имена требуемых файлов должны разделяться хотя бы одним пробелом. Если они не помещаются на одной строке, может быть специфицирован признак продолжения — символ \.

Может быть задано любое число команд DOS и/или имен исполнительных файлов, но каждая команда или имя должны располагаться на отдельной строке и начинаться с символа **TAB** или хотя бы одного пробела. Команды или файлы выполняются лишь в том случае, если хотя бы один из требуемых файлов был модифицирован после создания или модификации целевого, то есть, должно выполняться одно из двух условий:

- целевой файл старше требуемого;
- целевой файл не существует.

Может быть задано любое число описаний цели/источника. Последняя строка предыдущего описания должна отделяться от первой строки следующего описания хотя бы одной строкой, содержащей все пробелы.

При появлении символа # остаток строки считается комментарием. В области команд символ # может находиться только в 1-й позиции строки.

Следует помнить, что порядок следования описаний крайне важен, так как в процессе их отработки могут меняться даты модификаций файлов, что оказывает влияние на дальнейшую работу MAKE.

Запуск MAKE осуществляется введением командной строки следующего вида:

```
MAKE [<опции>][<макроопределения>]<имя файла описаний>
```

Имя файла описаний MAKE обычно не имеет расширения и совпадает с базовыми именами используемых в описаниях файлов, но это не является обязательным.

Если MAKE обнаруживает, что очередное описание по каким-либо причинам не может быть отработано, осуществляется переход к следующему описанию.

Если в процессе работы выяснится, что целевой файл не существует, MAKE продолжает работу, так как этот файл может быть создан следующими командами. Если же не существует требуемый или командный файл или возникает ошибка при выполнении команды, MAKE прекращает свою работу, а на консоль выдается поясняющее сообщение.

Опции MAKE

Каждая опция MAKE в командной строке запуска MAKE обозначается предшествующим символом /.

Описание опций MAKE приведено ниже:

/D

Выдавать на консоль даты последних модификаций каждого сканируемого файла.

/I

Игнорировать коды возврата после вызываемых программ.

/N

Выдавать на консоль команды, выполнение которых не осуществляется.

/S

Не выдавать на консоль сообщений.

Макроопределения

Использование макроопределений позволяет отложить определение компонент описания работы MAKE до момента запуска. Они могут располагаться как в файле описаний, так и в командной строке.

Существуют две формы макроопределений:

```
<имя>=<значение>
```

или

```
$(<имя>)
```

Первая форма задает значение символическому параметру, который может использоваться для определения компонент описания. Допустимо любое число пробелов между элементом **<имя>** и символом = и между этим символом и элементом **<значение>**, которые игнорируются. Пробелы, специфицированные после **<значение>**, рассматриваются как часть значения. Пробелы как часть значения в командной строке должны заключаться в двойные апострофы ".

В файле описаний MAKE каждое макроопределение должно занимать отдельную строку.

Одно и то же имя может быть определено в нескольких местах. Подстановка значений осуществляется в соответствии со следующим списком (в порядке убывания приоритетов):

- Из командой строки.
- Из файла описаний MAKE.
- Из текущего окружения (например, ключевые слова DOS).

Вторая форма макроопределений использует значение, определенное в другом месте. Элемент **<имя>** приводится к изображению на регистре заглавных букв.

Допускается вложенность макроопределений, когда внутреннее макроопределение определяется через внешнее. При этом следует избегать рекурсии.

Пример рекурсивной вложенности макроопределений:

```
A=$(B)
B=$(A)
```

Существуют 3 специальные макропеременные, имеющие следующие фиксированные значения:

- **$*** — часть имени (без расширения) целевого файла;
- **$@** — полное имя целевого файла;
- **$**** — полный список требуемых файлов.

Эти макропеременные не требуют предварительного описания и могут использоваться в файле описаний MAKE.

Правила вывода

MAKE обеспечивает возможность задания правил вывода, которые помогают правильно интерпретировать неполностью определенные конструкции.

Правила вывода могут находиться в файле описаний MAKE или в специальном файле с именем **TOOLS.INI**, поиск которого осуществляется на активном драйве в подоглавлениях, определенных командой DOS **PATH**. В файле **TOOLS.INI** правилам вывода должна предшествовать строка, первыми символами которой являются **[make]**.

Поиск правила вывода осуществляется в следующей последовательности:

- В файле описаний MAKE.
- В файле **TOOLS.INI**.

Правила вывода задаются в виде:

```
.<расширение требуемого файла>.<расширение целевого файла>:
<команда>
<команда>
...
```

Пример: содержимое файла описаний MAKE:

```
.asm.obj:
MASM $*.asm,,,; test1.obj: test1.asm test2.obj: test2.asm
MASM test2.asm;
```

Правило вывода занимает первые 2 строки. Прочитав 3-ю строку, MAKE обнаруживает, что описание неполно, так как 4-я строка является уже началом следующего описания. Поиск нужного правила вывода ведется по совпадению расширений файлов строки 3 с указанными в правиле. После отыскания правила MAKE, обработав макропеременную **$***, выполняет командную строку:

```
MASM test1.asm,,;
```

4-я и 5-я строки переставляют собой законченное описание, и для его интерпретации использования правил вывода не требуется.



Сегментация программы

Программа на языке Ассемблера состоит из последовательности программных сегментов, заканчивающейся директивой **END**. Начало каждого сегмента обозначается директивой **SEGMENT**, конец — директивой **ENDS**.

В каждом сегменте при помощи директивы **ASSUME** могут быть определены используемые по умолчанию для адресации элементов программы регистры сегмента.

В каждом сегменте могут быть выделены специальные программные единицы (процедуры), позволяющие использовать часть программного кода многократно без его дублирования в разных частях программы. Процедуры обычно включены в систему адресации сегмента. Начало и конец процедуры определяются директивами **PROC** и **ENDP** соответственно.

Сегменты могут быть объединены в группу при помощи директивы **GROUP**.

Директивы **ORG** и **EVEN** позволяют управлять адресами размещения инструкций процессора.

Директивы SEGMENT и ENDS

Синтаксис:

```
имя SEGMENT [[выравнивание]] [[комбинация]] [['класс']]
имя ENDS
```

Директивы **SEGMENT** и **ENDS** отмечают соответственно начало и конец программного сегмента и должны быть помечены одним и тем же именем, которое и считается именем сегмента. Программный сегмент представляет собой последовательность инструкций и/или полей данных, адресуемых относительно одного регистра сегмента. Имя сегмента должно быть уникальным и может появляться в поле метки только лишь в другом предложении **SEGMENT**. Все директивы **SEGMENT** с одним и тем же именем обозначают продолжение одного и того же сегмента. При этом следует помнить, что параметры всех директив **SEGMENT**, определяющих один и тот же программный сегмент, не должны противоречить друг другу.

Параметры выравнивания, комбинация и класс директивы **SEGMENT** задают информацию для линкера. Они должны кодироваться в указанной последовательности, но могут быть опущены в произвольной комбинации.

Выравнивание определяет границу адреса, начиная с которой сегмент будет загружаться в память. Могут быть заданы следующие значения:

- **BYTE** — использовать любую границу;
- **WORD** — граница слова (2 байта);
- **PARA** — граница параграфа (16 байтов);
- **PAGE** — граница страницы (256 байтов).

Если выравнивание не указано, предполагается **PARA**. Следует помнить, что точный адрес начала сегмента до его загрузки в память неизвестен. Тип выравнивания только накладывает на него ограничение.

Тип комбинации определяет возможность и способы объединения программных сегментов, имеющих одно имя. Могут быть указаны следующие значения:

- **PUBLIC** — все сегменты с одним и тем же именем объединяются в один непрерывный сегмент. Все инструкции и поля данных нового сегмента будут адресоваться относительно одного регистра сегмента, а все смещения будут вычисляться относительно начала этого сегмента.
- **STACK** — все сегменты с одним и тем же именем объединяются в один непрерывный сегмент. Этот тип комбинации отличается от **PUBLIC** лишь тем, что адресация в новом сегменте будет вестись относительно регистра **SS**; регистр **SP** при этом устанавливается на конец сегмента. Такой тип комбинации обычно имеют сегменты стека. Тип комбинации **STACK** автоматически обеспечивает инициализацию регистров **SS** и **SP**, и пользователю необязательно включать в свою программу инструкции для установки этих регистров.
- **COMMON** — все одноименные сегменты этого класса будут загружаться в память, начиная с одного адреса. Таким способом можно формировать оверлейные программы. Длина области загрузки равна длине максимального по объему сегмента. Все адреса в этих сегментах вычисляются относительно одного базового адреса. Если некоторые данные объявлены в более, чем одном сегменте с конкретным именем и типом комбинации **COMMON**, данные, объявленные последними, замещают все предыдущие.
- **MEMORY** — для Microsoft 8086 Object Linker (LINK) полностью совпадает с типом **PUBLIC**. **MASM** обеспечивает отдельный тип комбинации **MEMORY** для совместимости с программами **LINK**, различающими эти типы комбинации.
- **AT адрес** — все метки и адресные переменные сегмента должны быть вычислены относительно указанного адреса. Адрес может быть представлен любым допустимым

выражением, не содержащим ссылок вперед. Сегмент с этим типом комбинации обычно не содержит программного кода или инициализируемых данных, а включает в себя адресные значения, фиксированные для вычислительной машины (например, адрес буфера экрана).

Если тип комбинации не указан, сегмент ни с чем не объединяется и рассматривается как отдельная программная единица.

Класс сегмента определяет порядок следования сегментов в памяти. Сегменты одного класса загружаются в память один после другого до того, как начнут загружаться сегменты другого класса.

В качестве класса сегмента может быть указан любой идентификатор, на который распространяются все требования и ограничения языка Ассемблера (в том числе условия чувствительности к регистру). Поскольку класс сегмента рассматривается как идентификатор, он не может быть определен где-либо еще в программе.

Если класс не указан, LINK копирует сегменты в исполнительный файл в той последовательности, в которой они расположены в объектном файле. Эта последовательность сохраняется до тех пор, пока **LINK** не обнаружит 2 или более сегмента одного класса, после чего **LINK** начинает объединение сегментов. Сегменты одного класса копируются в последовательные блоки исполнительного файла.

Все сегменты имеют класс. Сегменты, для которых класс не указан, считаются принадлежащими к классу с пустым именем и копируются в последовательные блоки памяти вместе с такими же сегментами.

Число сегментов, принадлежащих к одному классу, не ограничено, но их суммарный объем не должен превышать 64K.

Если на вход программы **LINK** подается несколько объектных файлов, правильное кодирование классов сегментов вообще говоря не обеспечивает правильную последовательность сегментов в исполнительном файле, так как эта последовательность в этом случае зависит еще и от последовательности объектных файлов в командной строке.

Пусть, например, **LINK** обрабатывает 2 объектных файла, 1-й из которых содержит 2 сегмента с классами **CODE** и **STACK**, а 2-й — один сегмент класса **DATA**.

В исполнительном файле сегменты всегда будут расположены в последовательности **CODE**, **STACK**, **DATA**. Если, например, программисту необходимо, чтобы сегменты располагались в последовательности **CODE**, **DATA**, **STACK**, ему следует создать объектный файл, содержащий фиктивные сегменты с теми же именами и теми же классами, но расположенные в нужном ему порядке, и в командной строке запуска **LINK** указать его первым.

Исходная программа, соответствующая такому объектному файлу, может иметь следующий вид:

```
code SEGMENT PARA PUBLIC 'CODE'
code ENDS
data SEGMENT PARA PUBLIC 'DATA'
data ENDS
stack SEGMENT PARA STACK 'STACK'
stack ENDS
```

Этот прием не может быть использован для программ на языках C, FORTRAN, PASCAL и BASIC, так как компиляторы этих языков следуют определенным соглашениям о порядке сегментов, который не следует нарушать.

Другим способом управления последовательностью сегментов является кодирование опции **/A** MASM, которая предписывает MASM располагать сегменты в объектном файле в алфавитном порядке. Сочетание опции **/A** с формированием последовательности фиктивных сегментов позволяет реализовывать довольно сложные стратегии управления структурой исполнительного файла.

В некоторых ранних версиях MASM опция **/A** включена по умолчанию.

Директивы PROC и ENDP

Директивы **PROC** и **ENDP** служат для определения процедуры. Процедура представляет собой набор инструкций и директив, образующих некоторую подпрограмму в рамках какого-либо сегмента.

Процедура имеет следующий вид:

```
имя PROC [[расстояние]]
...
предложения
...
имя ENDP
```

Директивы **PROC** и **ENDP** обозначают соответственно начало и конец процедуры и должны быть помечены одним и тем же именем, которое считается именем процедуры.

Необязательное расстояние может принимать значения **FAR** и **NEAR**. Если этот параметр опущен, предполагается **NEAR**.

Имя процедуры имеет атрибуты метки и может быть использовано как операнд в инструкциях перехода, вызовах или циклах.

Возврат из процедуры должен быть выполнен инструкцией **RET**. При этом следует помнить, что адрес возврата выбирается из стека (в соответствии со значениями регистров **SS** и **SP**). Для процедур с расстоянием **NEAR** адрес возврата состоит только из смещения и занимает в стеке 2 байта. Для **FAR**-процедур он занимает 4 байта стека, включая в себя базовый адрес (содержимое регистра сегмента) и смещение.

Допускается вложенность процедур.

Процедуре могут быть переданы параметры. Вообще говоря, передача параметров и их распознавание в процедуре возлагается на программиста. Но при соблюдении стандартных соглашений, принятых в языках высокого уровня, параметры процедуры могут быть отслежены командой трассировки стека **K** SYMDEB.

Согласно стандартным соглашениям параметры размещаются в стеке, верх которого определяется содержимым регистров **SP** и **SS**.

Пример передачи параметров:

```
...
PUSH AX                     ; 2-й параметр
PUSH BX            ; 1-й параметр
CALL addup
ADD  SP,4          ; уничтожение параметров
...
addup PROC NEAR    ; адрес возврата для NEAR - 2 байта
PUSH BP            ; сохранение базового указателя
MOV  BP,SP         ; загрузка базового регистра
MOV  BX,[BP+4]     ; адрес 1-го параметра
MOV  AX,[BP+6]     ; адрес 2-го параметра
...
POP  BP
RET addup ENDP
```

Из этого примера ясно, что адрес возврата запоминается в верхушке стека перед параметрами (стек «растет» от больших адресов к малым).

Если бы процедура специфицировала расстояние **FAR**, адрес возврата занял бы 4 байта, а смещение для 1-го параметра составило бы 6 байтов.

Директива ASSUME

Синтаксис:

```
ASSUME регистр-сегмента:имя-сегмента...
ASSUME NOTHING
```

Директива **ASSUME** устанавливает регистр сегмента в качестве умалчиваемого регистра сегмента для адресации меток и переменных в указанном сегменте или группе. Последующие ссылки к метке или переменной при отсутствии явных указаний разрешаются относительно данного регистра сегмента.

В качестве регистра сегмента могут быть указаны **CS**, **DS**, **SS** или **ES**.

В качестве имени сегмента может быть специфицировано:

- Имя сегмента, предварительно определенное директивой **SEGMENT**.
- Имя группы, предварительно определенное директивой **GROUP**.
- Ключевое слово **NOTHING**.

Наличие ключевого слова **NOTHING** отменяет текущий выбор конкретного регистра сегмента или текущий выбор всех регистров сегмента (для второй формы директивы).

Выбор регистра сегмента по умолчанию в отдельном предложении языка Ассемблера может быть отменен при помощи оператора переключения сегмента (**:**).

Директива GROUP

Синтаксис:

```
имя GROUP имя-сегмента,...
```

Директива **GROUP** обозначает, что один или несколько сегментов с указанными именами логически объединяются в группу с данным именем, что позволяет адресовать все метки и переменные в этих сегментах относительно начала группы, а не начала содержащего их сегмента. Имя-сегмента должно быть именем сегмента, определенного директивой **SEGMENT**, или **SEG**-выражением. Оно должно быть уникальным.

Директива **GROUP** не влияет на порядок загрузки сегментов, который зависит от классов сегментов и их расположения в объектном файле.

Сегменты одной группы не обязательно будут занимать непрерывную область памяти. Они могут быть перемешаны с сегментами, не принадлежащими этой группе. Однако, расстояние в байтах между первым байтом первого сегмента группы и последним байтом последнего сегмента группы не должно превышать 64K. Таким образом, если сегменты группы расположены последовательно, группа может занимать до 64K оперативной памяти.

Имена групп могут использоваться в директиве **ASSUME** и в качестве префикса операнда оператора переключения сегмента (**:**).

Имя группы может появиться только в одной директиве **GROUP** в исходном файле. Если к группе принадлежат несколько сегментов в исходном файле, все их имена должны быть указаны в одной директиве **GROUP**.

Директива END

Синтаксис:

```
END [[выражение]]
```

Директива **END** обозначает конец модуля. Ассемблер игнорирует все предложения, следующие в исходном файле за этой директивой.

Необязательное выражение определяет точку входа программы, в которую будет передано управление при запуске программы на счет. Значением этого выражения должен быть адрес в одном из программных сегментов данного исходного файла.

Если выражение опущено, точка входа не определяется.

При попытке выполнения программы с незаданной точкой входа могут возникать ошибки, поэтому директиву **END** без параметров рекомендуется кодировать лишь с сегментами, содержащими только поля данных.

В исходном файле может быть определена только одна точка входа.

Директивы ORG и EVEN

Директивы **ORG** и **EVEN** позволяют задавать адрес памяти, начиная с которого будут располагаться последующие инструкции процессора.

Директива **ORG** имеет следующий формат:

```
ORG выражение
```


Значение указанного выражения присваивается указателю позиции, и адреса последующих инструкций и данных будут начинаться с нового значения.

Значением выражения должно быть абсолютное число, точнее, все используемые в нем имена должны быть известны на 1-м проходе Ассемблера. В качестве элемента выражения может быть использован знак указателя позиции (**$**), обозначающий его текущее значение.

Пример:

```
ORG 120h
MOV AX,BX
```

В этом примере инструкция **MOV** будет начинаться с байта 120h текущего сегмента.

Директива **EVEN** имеет следующий формат:

```
EVEN
```

Директива **EVEN** выравнивает следующее за ней поле данных или инструкцию по границе слова, то есть, по четному адресу. Если текущее значение указателя позиции нечетно, директива увеличивает его значение на 1 и генерирует инструкцию **NOP** (нет операции). Если текущее значение указателя позиции уже четно, никаких действий не производится.

Директива **EVEN** не должна использоваться в сегментах с типом выравнивания **BYTE**.



Условные директивы

Язык ассемблера включает в себя условные директивы двух типов: директивы условного ассемблирования и директивы условной генерации ошибок.

Директива условного ассемблирования обеспечивает ассемблирование блока предложений лишь в том случае, если истинно заданное в директиве условие.

Директива условной генерации ошибки проверяет заданное в ней условие и генерирует ошибку, то есть, формирует сообщение в листинге

программы и обеспечивает код возврата, соответствующий наличию ошибки в исходном тексте, если это условие истинно.

Директивы обоих типов проверяют условия времени ассемблирования. Они не могут анализировать условия времени выполнения, так как последние не могут быть вычислены до того, как программа начнет выполняться.

В условиях допустимы лишь выражения, преобразуемые в константы при ассемблировании.

Директивы условного ассемблирования

Директивы условного ассемблирования позволяют в некоторой степени управлять процессом ассемблирования. В зависимости от значений условий времени ассемблирования, проверяемых этими директивами, макроассемблер может пропустить обработку целого блока предложений или обработать вместо него другой (альтернативный) блок.

Блок предложений условного ассемблирования имеет следующий общий вид:

```
директива-условного-ассемблирования
..........
предложения Ассемблера или инструкции
..........
[[ ELSE ]]
..........
предложения Ассемблера или инструкции
..........
ENDIF
```

Директива-условного-ассемблирования задает условие, при истинном значении которого будут ассемблироваться предложения, расположенные непосредственно после директивы и до ключевого слова **ELSE**, или, если оно опущено, до конца блока, обозначаемого **ENDIF**. Если условие не выполняется, ассемблируется группа предложений, расположенная между ключевыми словами **ESLE** и **ENDIF**, а если **ELSE** опущено, MASM пропускает блок.

Допускается вложенность директив. Максимальная глубина вложения — 255.

В качестве **директивы-условного-ассемблирования** могут задаваться конструкции, приведенные ниже.

Директивы условного ассемблирования:

```
IF выражение
IFE выражение
IF1
IF2
IFDEF имя
IFNDEF имя
IFB <аргумент>
IFNB <аргумент>
IFIDN <аргумент-1>,<аргумент-2>
IFDIF <аргумент-2>,<аргумент-2>
```

В случае директивы **IF** блок ассемблируется, если указанное выражение истинно (не нуль). Для директивы **IFE** блок ассемблируется, если выражение ложно (нуль). Выражение должно иметь абсолютное значение и не может содержать ссылок вперед.

Директивы **IF1** и **IF2** проверяют номер прохода Ассемблера и обеспечивают обработку блока только на 1-м (**IF1**) или только на 2-м (**IF2**) проходе.

Пример (выдача сообщений на консоль):

```
IF1
%OUT Pass 1
ELSE
%OUT Pass 2
ENDIF
```

Директивы **IFDEF** и **IFNDEF** проверяют, определено ли в программе указанное имя. **IFDEF** обеспечивает ассемблирование, если это имя определено как метка, переменная или символ, **IFNDEF** — если имя не определено. Заметим, что если в качестве имени задана ссылка вперед, она считается неопределенной на 1-м проходе и определенной на 2-м.

Имена можно определять не только указанием их в соответствующей позиции при мнемонике инструкции или директивах определения памяти, но в опции Ассемблера **/D**, что позволяет управлять составом ассемблируемой программы без изменения исходного файла.

Директивы **IFB** и **IFNB** проверяют значение указанного аргумента, трактуемого как строка символов, и вызывают ассемблирование блока, если аргумент является пробелом (**IFB**) или отличен от пробела (**IFNB**). Аргумент может быть именем, числом или выражением. Скобки < и > обязательны.

Директивы **IFB** и **IFNB** введены для использования их в макроопределениях, где они могут управлять условным ассемблированием, основываясь на том, задан или нет параметр макроопределения. В этом случае в качестве аргумента следует задавать один из формальных параметров макроопределения.

Директивы **IFDIF** и **IFIDN** сравнивают специфицированные для них аргументы, которые трактуются как символьные строки (с учетом регистра), и вызывают ассемблирование блока при их идентичности (**IFIDN**) или неидентичности (**IFDIF**). Аргументы могут быть именами, числами или выражениями. Скобки < и > обязательны. Аргументы разделяются запятой.

Директивы **IFDIF** и **IFIDN** введены для использования их в макроопределениях, где они могут управлять условным ассемблированием, анализируя значения передаваемых в макроопределение параметров. В этом случае в качестве аргументов следует задавать формальные параметры макроопределения.



Директивы условной генерации ошибок

Использование директив условной генерации ошибок позволяет контролировать ситуации, не нашедшие отражения в синтаксисе языка Ассемблера.

Например, при помощи такой директивы можно вызвать генерацию сообщения об ошибке и, соответственно, установку минимального кода возврата транслятора в случае использования регистра AX, что с точки зрения синтаксиса не является ошибкой, но может быть недопустимо по логике программы.

Обработка любой директивы условной генерации ошибок, кроме **.ERR1**, с истинным условием эквивалентна распознаванию фатальной ошибки (с кодом возврата 7), при которой MASM уничтожает объектный файл.

Форматы директив и соответствующие им сообщения MASM приведены ниже.

Директивы условной генерации ошибок:

Синтаксис	Номер и текст сообщения
ERR1	87 Forced error - pass 1
ERR2	88 Forced error - pass 2
ERR	89 Forced error
ERRE выражение	90 Forced error - expression equals 0
ERRNZ выражение	91 Forced error - expression not equals 0
ERRNDEF имя	92 Forced error - symbol not defined
ERRDEF имя	93 Forced error - symbol defined
ERRB <строка>	94 Forced error - string blank
ERRNB <строка>	95 Forced error - string not blank
ERRIDN <строка-1>	96 Forced error - strings identical <строка-2>
ERRDIF <строка-2>	97 Forced error - strings different <строка-2>

Директива **.ERR** обеспечивает безусловную генерацию сообщения об ошибке. Директивы **.ERR1** и **.ERR2** также безусловно воздействуют соответственно лишь на 1-й и 2-й проход транслятора. Директива **.ERR1** вызывает не фатальную ошибку, а предупреждение.

Директивы **.ERRNZ** и **.ERRE** вычисляют значение указанного выражения и обеспечивают появление сообщения об ошибке соответственно в случае истинности (1) или ложности (0) этого выражения.

Директива **.ERRDEF** обеспечивает появление фатальной ошибки, когда указанное имя определено в программе как метка, переменная или символ, а директива **.ERRNDEF** — когда имя еще не определено. Если это имя является ссылкой вперед, оно считается неопределенным на 1-м проходе и определенным на 2-м. Эти директивы работают на 1-м проходе.

Директивы **.ERRB** и **.ERRNB** анализируют указанную строку и обеспечивают генерацию сообщения об ошибке, если эта строка соответственно содержит все пробелы или нет.

Директивы **.ERRIDN** и **.ERRDIF** вызывают появление фатальной ошибки, если указанные строки соответственно идентичны или различны.



Макросредства

Макросредства языка Ассемблера позволяют формировать в исходной программе блоки предложений (макроопределения), имеющие одно общее имя, и затем многократно использовать это имя для представления всего блока. В процессе ассемблирования MASM автоматически замещает каждое распознаваемое макроимя (макрокоманду) последовательностью предложений в соответствии с макроопределением, в результате чего формируется макрорасширение.

Макрокоманда может встречаться в исходной программе столько раз, сколько это необходимо. Макроопределение в исходном файле должно предшествовать его первому использованию в макрокоманде. Макроопределение может как непосредственно находиться в тексте программы, так и подключаться из другого файла директивой **INCLUDE**.

В макроопределение могут быть переданы параметры, которые могут управлять макроподстановкой или задавать фрагменты текста.

В программе на языке Ассемблера макрокоманды выполняют в целом те же функции, что и процедуры, то есть, обеспечивают многократное и функционально законченное действие с параметрическим управлением.

Различие заключается в следующем:

- Процедура присутствует в исходной программе один раз, тогда как тело макроопределения дублируется столько раз, сколько соответствующих макрокоманд содержит исходный файл.
- Текст процедуры статичен и неизменен в то время, как состав макрорасширения может зависеть от параметров макрокоманды. Следует помнить, что параметры макрокоманды — это значения времени ассемблирования, а параметры процедуры принимают какие-то определенные значения лишь в процессе выполнения программы.



Макродирективы

Макроопределение представляет собой блок исходных предложений, начинающийся директивой **MACRO** и заканчивающийся директивой **ENDM**. Формат макроопределения:

```
имя MACRO [[формальный-параметр,...]]
предложения
ENDM
```

Именем макроопределения считается имя, указанное в директиве **MACRO**. Оно должно быть уникальным и используется в исходном файле для вызова макроопределения. Формальные параметры представляют собой внутренние по отношению к данному макроопределению имена, которые используются для обозначения значений, передаваемых в макроопределение при его вызове. Может быть определено любое число формальных параметров, но все они должны помещаться на одной строке и разделяться запятыми (если их несколько).

В теле макроопределения допустимы любые предложения языка ассемблера, в том числе и другие макродирективы. Допустимо любое число предложений. Каждый формальный параметр может быть использован любое число раз в этих предложениях.

Макроопределения могут быть вложенными, то есть, одно макроопределение может содержаться в другом. MASM не обрабатывает вложенные макроопределения до тех пор, пока не будет вызвано внешнее макроопределение. Поэтому вложенное макроопределение не может быть вызвано до тех пор, пока хотя бы один раз не было вызвано внешнее макроопределение. Глубина вложенности может быть произвольной и ограничивается лишь размером доступной при ассемблировании памяти.

Макроопределения могут содержать вызовы других макроопределений. Такие вложенные макровызовы обрабатываются так же, как и другие макровызовы, но лишь тогда, когда вызвано внешнее макроопределение.

Макроопределения могут быть рекурсивными, то есть, вызывать сами себя.

MASM ассемблирует предложения тела макроопределения только при макровызове и только в той точке исходного файла, где этот вызов имеет место. Таким образом, все адреса в ассемблируемом коде будут связаны с точкой макровызова, а не местоположением макроопределения. Макроопределение само по себе никогда не ассемблируется.

Следует соблюдать осторожность при использовании слова **MACRO** после директив **TITLE**, **SUBTTL** и **NAME**. Так как директива **MACRO** «перекрывает» эти директивы, использование такого слова после указанных директив приведет к тому, что Ассемблер начнет создавать макро с именами **TITLE**, **SUBTTL** или **NAME**. Например, предложение

```
TITLE Macro File
```

может быть внесено в исходный файл для создания заголовка «Macro File», но в результате будет создано макроопределение с именем **TITLE**, имеющее формальный параметр **File**, а, поскольку в этом случае, очевидно, не будет предусмотрена директива **ENDM**, закрывающая макроопределение, будет скорее всего генерироваться сообщение об ошибке.

Следует помнить, что MASM замещает все вхождения имени формального параметра в теле макроопределения его фактическим значением, даже если программисту это не нужно. Например, если формальный параметр имеет имя **AX**, в генерируемом макрорасширении MASM замещает все вхождения **AX** на значение фактического параметра. Если же по логике макроопределения необходимо использование регистра **AX**, а не одноименного формального параметра, код макрорасширения может быть некорректным.

Макроопределения могут быть переопределены. При этом можно не заботиться об удалении из исходного файла первого макроопределения, так как каждое следующее макроопределение автоматически замещает предыдущее макроопределение с тем же именем. Если переопределение совершается внутри самого макроопределения, необходимо помнить, что между директивами **ENDM**, закрывающими старое и новое макроопределения, не должно находиться никаких строк. Например, следующее переопределение некорректно:

```
dostuff MACRO
...
dostuf MACRO
...
ENDM
;; Ошибочная строка
ENDM
```


Макроопределение может быть вызвано в любой момент простым указанием его имени в исходном файле (имена макро в комментариях игнорируются). MASM при этом копирует предложения макроопределения в точку вызова, замещая в этих предложениях формальные параметры на фактические параметры, задаваемые при вызове.

Общий вид макровызова:

```
имя [[фактический-параметр,...]]
```

Имя должно быть именем ранее определенного в исходном файле макроопределения. Фактическим параметром может быть имя, число или другое значение. Допустимо любое число фактических параметров, но все они должны помещаться на одной строке. Элементы списка параметров должны разделяться запятыми, пробелами, или TAB-символами.

MASM замещает первый формальный параметр на первый фактический параметр, второй формальный параметр на второй фактический параметр. Если фактических параметров в макровызове больше, чем формальных параметров в макроопределении, лишние фактические параметры игнорируются. Если же фактических параметров меньше, чем формальных, формальные параметры, для которых не заданы фактические, замещаются пустыми строками (пробелами). Для определения того, заданы или не заданы соответствующие фактические параметры могут быть использованы директивы **IFB**, **IFNB**, **.ERRB** и **.ERRNB**.

В макросредствах языка Ассемблера имеется возможность передавать список значений в качестве одного параметра. Этот список должен быть заключен в одинарные скобки < и >, а сами элементы списка — разделяться запятыми. Пример:

```
allocb <1,2,3,4>
```

При написании макроопределений иногда возникает необходимость в задании меток инструкций или имен полей данных. Поскольку каждое макроопределение может использоваться многократно, может возникнуть ситуация, когда несколько имен или меток определены многократно, что вызовет ошибку трансляции. Для обеспечения правильной обработки таких ситуаций в макроязыке предусмотрена директива **LOCAL**, позволяющая локализовать заданные имена исключительно в данном макрорасширении.

Формат:

```
LOCAL формальное-имя,...
```

Формальное-имя может затем использоваться в данном макроопределении с уверенностью, что при каждом макровызове его значение будет уникальным. В директиве **LOCAL**, если она присутствует, должно

быть задано хотя бы одно имя, а если их несколько, они должны разделяться запятыми.

Для обеспечения уникальности определенных директивой **LOCAL** имен MASM для каждого такого имени при каждом макровызове порождает реальное имя следующего вида:

```
??номер
```

Номер представляет собой 16-ричное число в пределах от 0000 до FFFF. Для предотвращения повторного определения имен программисту не рекомендуется определять в своей программе имена этого типа.

Директива **LOCAL** может использоваться только в макроопределении, причем, там она должна предшествовать всем другим предложениям макроопределения, то есть, следовать непосредственно после **MACRO**.

Для удаления макроопределений служит директива **PURGE**.

Формат:

```
PURGE имя-макроопределения,...
```

Удаляются все текущие макроопределения с указанными именами. Последующий вызов одного из этих макроопределений будет приводить к ошибке.

Директива **PURGE** введена для возможности освобождения и повторного использования памяти, занимаемой неиспользуемыми в дальнейшем макроопределениями. Если **имя-макроопределения** представляет мнемонику инструкции или директивы, восстанавливается первоначальный смысл мнемоники в соответствии со значением данного ключевого слова.

Директива **PURGE** часто используется для удаления ненужных макроопределений из подключаемой директивой **INCLUDE** библиотеки макроопределений. Библиотека макроопределений представляет собой обычный последовательный файл, который в общем случае может содержать большое число макроопределений. Комбинация директив **INCLUDE** и **PURGE** позволяет выбрать из них только нужные для данной программы, что сократит размер исходного файла.

Нет необходимости удалять макроопределения после их переопределения, так как каждое переопределение автоматически удаляет предшествующее макроопределение с данным именем. Аналогично каждое макроопределение может удалить само себя, имея в своей последней обрабатываемой строке директиву **PURGE**.

Конец текущего макроопределения обозначается директивой **ENDM**, которая должна находиться в последней строке макроопределения.

Формат:

```
ENDM
```

Выход из текущего макроопределения до достижения директивы **ENDM** обеспечивается директивой **EXITM**, имеющей следующий формат:

```
EXITM
```

Выход из макроопределения по директивам **ENDM** и **EXITM** заключается в прекращении генерации текущего макрорасширения и возврате в точку вызова текущего макроопределения в динамически внешнем макрорасширении или в исходной программе.

Пример:

```
add   MACRO param
IFB   param
EXITM
ENDIF
ADD   AX,param
ENDM
```

В этом мароопределении осуществляется добавление величины, определяемой формальным параметром **param**, к содержимому регистра **AX**. Блок условного ассемблирования **IFB** обеспечивает выход из макроопределения, если при вызове параметр не был задан.



Блоки повторений

Блок повторения представляет собой специфическую форму макроопределения с несколько ограниченными возможностями макроязыка. По сути блок повторения представляет собой макрообъект, объединяющий в себе макроопределение и макровызов. Макроопределения как шаблона, по которому производится генерация макрорасширения, здесь не требуется, в результате чего сужаются возможности варьирования генерируемым текстом. С другой стороны, использование блоков повторения по сравнению с макроопределениями является синтаксически и ло-

гически более простой задачей. Обработка блока повторения заключается в многократном дублировании тела блока с незначительными изменениями текста.

В языке Ассемблера имеются блоки повторения 3-х типов:

- **REPT**-блок;
- **IRP**-блок;
- **IRPC**-блок.

REPT-блок имеет следующий формат:

```
REPT выражение
...
предложения
...
ENDM
```

Блок предложений, заключенный между ключевыми словами **REPT** и **ENDM**, повторяется столько раз, каково текущее значение указанного выражения. Выражение должно иметь значение в виде 16-битового числа без знака и не может содержать внешних или неопределенных символов. Тело блока может включать в себя любые предложения языка.

IRP-блок имеет следующий формат:

```
IPR формальный-параметр,<параметр,...>
...
предложения
...
ENDM
```

Блок предложений, стоящий между ключевыми словами **IRP** и **ENDM**, будет повторен для каждого параметра в списке, заключенном в скобках <>.

Формальный-параметр относится только к данному блоку и последовательно принимает значения из списка. В качестве параметров в списке могут быть заданы определенные символы, строки, числа или символьные константы. Может быть задано любое число параметров, которые, если их несколько, должны разделяться запятыми. Тело блока может включать в себя любые предложения языка Ассемблера. **Формальный-параметр** в теле блока может быть использован произвольное число раз.

Когда MASM распознает директиву **IRP**, он создает копию предложений блока для каждого параметра в списке. При копировании предложений осуществляется замена текущим параметром всех вхождений

формального параметра блока. Если в списке будет обнаружен пустой параметр (<>), формальный параметр получит значение пустой строки. Если список параметров пуст, блок игнорируется.

Пример:

```
alloc MACRO x
IRP   y,<x>
DB    y
ENDM
ENDM
```

В результате обработки макровызова

```
alloc <0,1,2,3>
```

будет сгенерировано макрорасширение

```
IRP   y,<0,1,2,3>
DB    y
ENDM
```

IRPC-блок имеет следующий формат:

```
IRPC формальный-параметр,строка
...
предложения
...
ENDM
```

Блок предложений, заключенный между ключевыми словами **IRPC** и **ENDM**, будет повторен для каждого символа указанной строки. При этом каждое вхождение формального параметра в блоке замещается текущим символом строки.

Строка представляет собой любую комбинацию букв, цифр и других символов. Если строка содержит пробелы, запятые или другие ограничители, она должна быть заключена в одинарные скобки <>.

Тело блока может включать в себя любые предложения языка Ассемблера. Формальный параметр в теле блока может быть использован произвольное число раз.

Когда MASM распознает директиву **IRPC**, он создает копию предложений блока для каждого символа в строке. При копировании предложений осуществляется замена текущим символом строки всех вхождений формального параметра блока.



Макрооператоры

В качестве элементов генерируемого текста в теле макроопределения могут быть использованы макрооператоры.

Оператор замены **&** обеспечивает замену в тексте формального параметра на значение соответствующего ему фактического параметра макроопределения. Символ **&** употребляется как перед, так и после имени формального параметра и служит для выделения параметра в строке символов или в строке, содержащей кавычки.

Во вложенных макроопределениях оператор замены может быть использован для задержки замены формального параметра действительным значением, для чего используется более общая форма оператора замены, в которой символ **&** может быть указан несколько раз. MASM трактует количество рядом стоящих символов **&** как глубину вложенности макроопределений (относительно текущего уровня), на которой следует осуществлять замену формального параметра.

Пример:

```
alloc MACRO x
IRP z,<1,2,3> x&&z DB z
ENDM
ENDM
```

В этом примере замена формального параметра **x** осуществляется немедленно при вызове макроопределения. Замена параметра **z** будет задержана до начала обработки блока повторения **IRP** с тем, чтобы параметр **z** получал значения из указанного списка. Таким образом, параметр **z** будет замещаться значением элемента списка каждый раз при переходе к следующему элементу списка блока повторения. В итоге по макрокоманде

```
alloc var
```

будет сгенерировано макрорасширение

```
var1  DB    1
var2  DB    2
var3  DB    3
```

Текстовый оператор литерала указывает MASM, что заданный внутри скобок < и > текст следует трактовать как простой литеральный

элемент независимо от того, содержит ли он запятые, пробелы или другие ограничители. Чаще всего этот оператор используется в макровызовах и директивах **IRP** для того, чтобы значения списка параметров обрабатывались как один параметр.

Кроме того, этот оператор используется в тех случаях, когда необходимо, чтобы MASM трактовал некоторые специальные символы (например, **;** или **&**) как литералы. Например, в выражении **<;>** точка с запятой рассматривается как точка с запятой, а не как начало комментария.

Символьный оператор литерала **!** отличается от текстового лишь тем, что в нем в качестве литерала рассматривается только непосредственно следующий за **!** символ. Например, выражения **!;** и **<;>** эквивалентны.

Оператор выражения **%** указывает MASM, что данный текст следует трактовать как выражение. MASM вычисляет значение выражения с учетом основания системы счисления встречающихся в выражении чисел и замещает текст этим значением. Текст должен представлять корректное выражение.

Оператор выражения обычно используется в макровызовах, когда в макроопределение необходимо передать значение выражения, а не его текстовую форму.

Пример:

```
area  MACRO par
DW    &par
ENDM
sum1  EQU   100
sum2  EQU   200
```

В результате обработки макровызова

```
area %(sum1+sum2)
```

будет сгенерировано макрорасширение

```
DW    300
```

Оператор макрокомментария **;;** определяет, что остаток строки является макрокомментарием.

В отличие от обычного комментария, который обозначается одинарными точкой с запятой и также может встречаться в макроопределениях, макрокомментарий в текст макрорасширения не попадает. Он присутствует лишь в теле макроопределения.



Директивы определения памяти

Директивы определения памяти служат для задания размеров, содержимого и местоположения полей данных, используемых в программе на языке Ассемблера. В отличие от других директив языка Ассемблера при обработке директив определения памяти генерируется объектный код. MASM транслирует задаваемые в директивах определения памяти числа, строки и выражения в отдельные образы байтов, слов или других единиц данных. Эти образы копируются в объектный файл.

Директивы определения данных могут задавать:

- Скалярные данные, представляющие собой единичное значение или набор единичных значений.
- Записи, позволяющие манипулировать с данными на уровне бит.
- Структуры, отражающие некоторую логическую структуру данных.



Скалярные данные

Директива DB обеспечивает распределение и инициализацию 1 байта памяти для каждого из указанных значений. В качестве значения может кодироваться целое число, строковая константа, оператор DUP, абсолютное выражение или знак **?**. Знак **?** обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми.

Если директива имеет имя, создается переменная типа **BYTE** с соответствующим данному значению указателя позиции смещением.

Строковая константа может содержать столько символов, сколько помещается на одной строке.

Символы строки хранятся в памяти в порядке их следования, то есть, 1-й символ имеет самый младший адрес, последний — самый старший.

Директива DW обеспечивает распределение и инициализацию слова памяти (2 байта) для каждого из указанных значений. В качестве значения может кодироваться целое число, 1- или 2-х символьная константа, оператор **DUP**, абсолютное выражение, адресное выражение или знак **?**. Знак **?** обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми.

Если директива имеет имя, создается переменная типа **WORD** с соответствующим данному значению указателя позиции смещением.

Строковая константа не может содержать более 2-х символов. Последний (или единственный) символ строки хранится в младшем байте слова. Старший байт содержит первый символ или, если строка односимвольная, 0.

Директива **DD** обеспечивает распределение и инициализацию двойного слова памяти (4 байта) для каждого из указанных значений. В качестве значения может кодироваться целое число, 1- или 2-х символьная константа, действительное число, кодированное действительное число, оператор **DUP**, абсолютное выражение, адресное выражение или знак **?**.

Знак **?** обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми.

Если директива имеет имя, создается переменная типа **DWORD** с соответствующим данному значению указателя позиции смещением.

Строковая константа не может содержать более 2-х символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, 0. Остальные байты заполняются нулями.

Директива **DQ** обеспечивает распределение и инициализацию 8 байтов памяти для каждого из указанных значений. В качестве значения может кодироваться целое число, 1- или 2-х символьная константа, действительное число, кодированное действительное число, оператор **DUP**, абсолютное выражение, адресное выражение или знак **?**.

Знак **?** обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми.

Если директива имеет имя, создается переменная типа **QWORD** с соответствующим данному значению указателя позиции смещением.

Строковая константа не может содержать более 2-х символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, 0. Остальные байты заполняются нулями.

Директива **DT** обеспечивает распределение и инициализацию 10 байтов памяти для каждого из указанных значений. В качестве значения может кодироваться целое выражение, упакованное десятичное число, 1- или 2-х символьная константа, кодированное действительное число, оператор **DUP** или знак ?. Знак ? обозначает неопределенное значение. Значения, если их несколько, должны разделяться запятыми.

Если директива имеет имя, создается переменная типа **TWORD** с соответствующим данному значению указателя позиции смещением.

Строковая константа не может содержать более 2-х символов. Последний (или единственный) символ строки хранится в младшем байте слова. Второй байт содержит первый символ или, если строка односимвольная, 0. Остальные байты заполняются нулями.

При обработке директивы **DT** подразумевается, что константы, содержащие десятичные цифры, представляют собой не целые, а десятичные упакованные числа. В случае необходимости определить 10-байтовое целое число следует после числа указать спецификатор системы счисления (**D** или**d** для десятичных чисел, **H** или **h** для 16-ричных).

Если в одной директиве определения памяти заданы несколько значений, им распределяются последовательные байты памяти.

Во всех директивах определения памяти в качестве одного из значений может быть задан оператор **DUP**.

Он имеет следующий формат:

```
счетчик DUP (значение,...)
```

Указанный в скобках **()** список значений повторяется многократно в соответствии со значением счетчика. Каждое значение в скобках может быть любым выражением, имеющим значением целое число, символьную константу или другой оператор **DUP** (допускается до 17 уровней вложенности операторов **DUP**). Значения, если их несколько, должны разделяться запятыми.

Оператор **DUP** может использоваться не только при определении памяти, но и в других директивах.

Примеры директив определения скалярных данных:

```
integer1  DB      16
string1   DB      'abCDf'
```



```
empty1    DB      ?
integer2  DW      16728
contan2   DW      4*3
multip2   DW      1,'$'
arr2      DW      array
string3   DD      'ab'
real3     DD      1.5
encode3   DD      3F000000r
high3     DD      4294967295
charu     DQ      'b'
encodeu   DQ      3F00000000000000r
highu     DQ      18446744073709551615
pack5     DT      1234567890
real5     DT      1.5
mult5     DT      'a',3F000000000000000000r
high5     DT      1208925819614629174706175d
db6               DB      5 DUP(5 DUP(5 DUP(10)))
dw6               DW      DUP(1,2,3,4,5)
```



Записи

Запись представляет собой набор полей бит, объединенных одним именем. Каждое поле записи имеет собственную длину, исчисляемую в битах, и не обязана занимать целое число байтов.

Объявление записи в программе на языке Ассемблера включает в себя 2 действия:

1. Объявление шаблона или типа записи директивой **RECORD**.

2. Объявление собственно записи.

Формат директивы **RECORD**:

```
имя-записи RECORD
имя-поля:длина[[=выражение]],...
```

Директива **RECORD** определяет вид 8- или 16-битовой записи, содержащей одно или несколько полей.

Имя-записи представляет собой имя типа записи, которое будет использоваться при объявлении записи. **Имя-поля** и длина (в битах) опи-

сывают конкретное поле записи. **Выражение**, если оно указано задает начальное (умалчиваемое) значение поля.

Описания полей записи в директиве **RECORD**, если их несколько, должны разделяться запятыми. Для одной записи может быть задано любое число полей, но их суммарная длина не должна превышать 16 бит.

Длина каждого поля задается константой в пределах от 1 до 16. Если общая длина полей превышает 8 бит, Ассемблер выделяет под запись 2 байта, в противном случае — 1 байт.

Если задано выражение, оно определяет начальное значение поля. Если длина поля не меньше 7 бит, в качестве выражения может быть использован символ в коде ASCII. Выражение не должно содержать ссылок вперед.

Внутри записи поля размещаются слева направо в порядке описания в директиве **RECORD**. Поэтому объявленное первым поле содержит самые значимые биты записи, если рассматривать ее как двоичное число. Если суммарная длина полей не равна 8 или 16, запись расширяется до целой границы байта нулевыми битами слева, и, таким образом, последний бит последнего поля всегда является самым младшим битом записи.

При обработке директивы **RECORD** формируется шаблон записи, а сами данные создаются при объявлении записи, которое имеет следующий вид:

```
[[имя]] имя-записи <[[значение,...]]>
```

По такому объявлению создается переменная типа записи с 8- или 16-битовым значением и структурой полей, соответствующей шаблону, заданному директивой **RECORD** с именем **имя-записи**.

Имя задает имя переменной типа записи. Если имя опущено, MASM распределяет память, но не создает переменную, которую можно было бы использовать для доступа к записи.

В скобках <> указывается список значений полей записи. Значения в списке, если их несколько, должны разделяться запятыми. Каждое значение может быть целым числом, строковой константой или выражением и должно соответствовать длине данного поля. Для каждого поля может быть задано одно значение.

Скобки <> обязательны, даже если начальные значения не заданы.

Если в качестве значения используется оператор **DUP**, в скобки <> следует заключать список значений оператора **DUP**. Следует иметь ввиду, что при использовании оператора **DUP** создается несколько переменных типа запись.

Задавать значения всех полей записи необязательно. Если Ассемблер вместо значения обнаружит левый пробел, будет использовано начальное значение поля, заданное директивой **RECORD**, а если оно и там опущено, значение поля будет не определено.



Структуры

Структура представляет собой набор полей байтов, объединенных одним именем.

Объявление структуры в программе на языке Ассемблера включает в себя 2 действия:

1. Объявление шаблона или типа структуры директивами **STRUC** и **ENDS**.

2. Объявление собственно структуры.

Формат объявления типа структуры:

```
имя STRUC
описания-полей
имя ENDS
```

Директивы **STRUC** и **ENDS** обозначают соответственно начало и конец описания шаблона (типа) структуры. Описание типа структуры задает имя типа структуры и число, типы и начальные значения полей структуры.

Указанное в директивах **STRUC** и **ENDS** имя задает новое имя типа структуры. Оно должно быть уникальным. **Описания-полей** определяют поля структуры и могут быть заданы в одной из следующих форм:

```
[[имя]] DB значение,...
[[имя]] DW значение,...
[[имя]] DD значение,...
[[имя]] DQ значение,...
[[имя]] DT значение,...
```

Каждое поле может иметь свое имя. Директивы **DB**, **DW**, **DD**, **DQ** или **DT** задают длину поля. Для каждого поля могут быть указаны начальные значения, которыми инициализируются поля при отсутствии соответствующих начальных значений при объявлении структуры. Имя каждой директивы, если оно задано, должно быть уникальным и представляет смещение поля относительно начала структуры.

Значением поля может быть число, символ, строковая константа или имя другого объекта. Для определения множества значений поля может использоваться оператор **DUP**. Если в качестве значения задана строковая константа, поле занимает столько байтов, сколько символов в константе. Если задано несколько значений, они должны разделяться запятыми.

Объявление типа структуры может содержать только описания полей и комментарии. По этой причине структуры не могут быть вложенными.

Пример:

```
table STRUC
count DB 10
value DW 10 DUP(?)
tname DB 'font'
table ENDS
```

При обработке директив **STRUC** и **ENDS** формируется шаблон структуры, а сами данные создаются при объявлении структуры, которое имеет следующий вид:

```
[[имя]] имя-структуры <[[значение,...]]>
```

По такому объявлению создается переменная типа структура со структурой полей, соответствующей шаблону, заданному директивой **STRUC** с именем **имя-структуры**.

Имя задает имя переменной. Если имя опущено, MASM распределяет память, но не создает переменную, которую можно было бы использовать для доступа к структуре.

В скобках <> указывается список значений полей структуры. Значения в списке, если их несколько, должны разделяться запятыми. Каждое значение может быть целым числом, строковой константой или выражением, тип которого должен совпадать с типом соответствующего ему поля. Для каждого поля может быть задано одно значение.

Скобки <> обязательны, даже если начальные значения не заданы.

Если в качестве значения используется оператор **DUP**, в скобки <> следует заключать список значений оператора **DUP**.

Задавать значения всех полей структуры необязательно. Если Ассемблер вместо значения обнаружит левый пробел, будет использовано начальное значение поля, заданное при описании типа структуры, а если и оно опущено, значение поля будет не определено.

Следует помнить, что объявлении структуры нельзя задавать значения полей, для которых в соответствующем шаблоне задано множество значений. Например:

```
strings STRUC
buffer DB 100 DUP(?)
crif   DB 13,10
guery  DB 'Filename'
endm   DB 35
strings ENDS
```

При объявлении структуры с использованием этого шаблона значения полей **buffer** и **crif** не могут быть заданы, так как шаблон для них определяет множество значений. Значение поля **guery** может быть перекрыто только значением, длина которого не превышает 8 байтов. Аналогично, значение поля **endm** может быть перекрыто любым однобайтовым значением.



Описание символических имен

Директивы описания символических имен позволяют создавать в исходной программе имена, использование которых существенно упрощает программирование. Символические имена являются константами времени ассемблирования и могут представлять число, текст, инструкцию или адрес.

Для описания символических имен в языке ассемблера служат директивы **EQU**, **LABEL** и директива абсолютного присваивания (**=**).

Директива абсолютного присваивания имеет следующий формат:

```
имя=выражение
```

По этой директиве создается абсолютное имя, представляющее значение, равное текущему значению указанного выражения. Для хранения этого значения не выделяется никакой памяти. Вместо этого каждое вхождение указанного имени в исходном файле замещается значением выражения.

Абсолютное имя может быть переопределено. В каждой директиве абсолютного присваивания в качестве имени может быть указано уникальное имя или имя, ранее использованное другой директивой абсолютного присваивания.

Выражение может быть целым числом, одно- или двухсимвольной строковой константой, константным выражением или адресным выражением. Его значение не должно превышать 65535.

Следует помнить, что значение абсолютного имени является переменной величиной в процессе ассемблирования и константой во время выполнения программы.

Примеры:

```
integer = 167
string  = 'ab'
const   = 3*4
addr    = string
```

Директива **EQU** имеет следующий формат:

```
имя EQU выражение
```

Директива **EQU** создает абсолютное имя, алиас или текстовое имя путем присваивания имени указанного выражения.

Под абсолютным здесь понимается имя, представляющее 16-битовое значение; алиасом называется ссылка на другое имя; текстовое имя представляет собой строку символов. Каждое вхождение имени в исходном файле Ассемблер замещает текстом или значением выражения в зависимости от типа используемого выражения.

Имя должно быть уникальным и не может быть переопределено. В качестве выражения может задаваться целое число, строковая константа, действительное число, кодированное действительное число, мнемоника инструкции, константное выражение или адресное выражение. Выражение, имеющее значением целое число в пределах от 0 до 65535, порождает абсолютное имя, вхождения которого Ассемблер замещает этим значением. Для всех остальных выражений вхождения имени замещаются текстом.

Директива **EQU** иногда используется для создания простых макроопределений.

Отметим, что замещение имен текстом или значением осуществляется до ассемблирования содержащего имя предложения.

Примеры:

```
k      EQU   102u       ;   значение
pi     EQU   3.1u       ;   текст
mat    EQU   20*30      ;   значение
adr    EQU   [BP]       ;   текст
cle    EQU   XOR AX,AX  ;   текст
d      EQU   BYTE PTR;      текст
t      EQU   'File'     ;   текст
```

Директива **LABEL** имеет следующий формат:

```
имя LABEL тип
```

Директива **LABEL** порождает новую переменную или метку путем присваивания имени указанного типа и текущего значения указателя позиции.

Имя должно быть уникальным и не может быть переопределено. В качестве типа может быть задано одно из следующих ключевых слов, имеющих обычный смысл:

- **BYTE**
- **WORD**
- **DWORD**
- **QWORD**
- **TBYTE**
- **NEAR**
- **FAR**.

Пример:

```
barray LABEL BYTE
warray DW    100 DUP(0)
```

Здесь имена **barray** и **warray** ссылаются на одну и ту же область памяти.



Директивы управления файлами

Директивы управления файлами позволяют управлять исходным и объектным файлами, а также листингом ассемблерной программы. Под управлением понимается указания о том, как трактовать элементы входного (исходного) файла и задание содержимого и объема вывода для выходных (объектный файл и листинг) файлов.

Управление исходным файлом

Для управления исходным файлом предназначены директивы **INCLUDE**, **.RADIX** и **COMMENT**.

Директива **INCLUDE** имеет следующий формат:

```
INCLUDE имя-файла
```

Содержимого файла с указанным именем, трактуемое как текст, вставляется в исходный файл на место директивы **INCLUDE**. **Имя-файла** должно определять существующий файл. **Имя-файла** может включать в себя полную или частичную информацию о пути поиска файла. Если **имя-файла** не содержит информацию о местонахождении файла, поиск осуществляется в директориях, заданных опцией **/I** MASM, а если файл там не будет найден, то — в текущей директории. Если файл не найден, MASM выдает сообщение об ошибке.

Когда Ассемблер обнаруживает директиву **INCLUDE**, он открывает указанный исходный файл и начинает ассемблировать содержащиеся в нем предложения. После обработки всех предложений этого файла Ассемблер продолжает ассемблирование с предложения, непосредственно следующего за **INCLUDE**.

Директивы **INCLUDE** могут быть вложенными. Файл, подключаемый по этой директиве, также может содержать директивы **INCLUDE**.

В листинге предложения из подключаемых файлов помечаются символом **C**.

При спецификации путей поиска файла могут использоваться символы / или \, что введено для совместимости с XENIX.

Если необходимо, чтобы местоположение подключаемых файлов задавалось динамически после формирования исходного файла, следует

в директивах **INCLUDE** опустить спецификацию путей поиска и определять их опцией **/I** или установкой текущего директория.

Примеры:

```
INCLUDE entry
INCLUDE b:\include\record
INCLUDE /include/as/stdio
INCLUDE local\define.inc
```

Директива **.RADIX** устанавливает умалчиваемое основание чисел во входном файле и имеет формат:

```
.RADIX выражение
```

В качестве выражения задается число в пределах от 2 до 16, которое определяет, в какой системе счисления трактовать числа при отсутствии явного указания. Могут быть указаны следующие значения:

- 2 — двоичная;
- 8 — 8-ричная;
- 10 — десятичная;
- 16 — 16-ричная.

Указанное в директиве **.RADIX** выражение всегда трактуется как десятичное число независимо от текущего умалчиваемого основания. По умолчанию при отсутствии директивы **.RADIX** используется десятичная система счисления.

Директива **.RADIX** не влияет на числа, указанные в директивах **DD**, **DQ** и **DT**. Числа выражений этих директив всегда трактуются как десятичные.

Директива **.RADIX** не влияет на спецификаторы основания **B** и **D**, используемые при задании целых чисел. Если последним символом целого числа оказывается **B** или **D**, он трактуется как спецификатор системы счисления, а не как 16-ричная цифра, даже если умалчиваемое основание — 16. Например, число 11B будет восприниматься как двоичное 11, а не как 16-ричное 11B. Для того, чтобы оно трактовалось как 16-ричное, следует задавать 11Bh.

Примеры:

```
.RADIX 16
.RADIX 2
```

Директива **COMMENT** позволяет указывать Ассемблеру, что выделенный участок исходного файла следует рассматривать как комментарий.

Формат:

```
COMMENT ограничитель
...
текст
...
ограничитель [[текст]]
```

Текст, заключенный между ограничителями (которые должны совпадать), считается комментарием и ассемблированию не подлежит. В качестве ограничителя берется первый отличный от пробела символ после ключевого слова **COMMENT**. Ассемблер пропускает весь последующий текст до следующего вхождения такого же ограничителя. Текст комментария не должен содержать такого символа. Комментарием считается также весь текст, расположенный на той же строке, что и последний ограничитель.

Директива **COMMENT** обычно используется при задании комментария, занимающего много строк.

Пример:

```
COMMENT
* Весь текст между звездочками считается комментарием *
```



Управление листингом

Директивы управления листингом позволяют управлять содержимым и форматом формируемого MASM листинга ассемблерной программы.

Директива **TITLE** задает заголовок листинга программы, который будет печататься в начале каждой страницы листинга. Текст заголовка может включать в себя до 60 любых символов. Каждый модуль может содержать не более одной директивы **TITLE**. Если не использовалась директива **NAME**, первые 6 отличных от пробела символов заданного в **TITLE** заголовка рассматриваются как имя модуля.

Директива **SUBTTL** определяет подзаголовок листинга, который печатается в начале каждой страницы листинга на следующей строке после заголовка. Задаваемый текст представляет собой любую комбинацию символов, из которой в качестве подзаголовка используются первые 60

символов. Если текст опущен, строка подзаголовка в листинге содержит левый пробел. Программа может содержать любое количество директив **SUBTTL**. Каждая последующая директива замещает предыдущую.

Директивы управления листингом

Формат	Функция
TITLE текст	задание заголовка листинга
SUBTTL [[текст]]	задание подзаголовка листинга
PAGE длина,ширина	задание параметров страницы листинга
.LIST	печатать листинг
.XLIST	не печатать листинг
.LALL	печатать все исходные предложения
.SALL	подавить печать макрорасширений
.XALL	печатать только код и данные
.SFCOND	подавить печать условных блоков с ложными условиями
.LFCOND	печатать условные блоки с ложными условиями
.TFCOND	установить умалчиваемый режим печати условных блоков
.CREF	печатать листинг перекрестных ссылок
.XCREF [[имя,...]]	подавить печать листинга перекрестных ссылок

Директива **PAGE** позволяет управлять форматом страницы листинга.

В первой форме директивы предусмотрены 2 позиционных параметра, которые устанавливают максимальное число строк страницы листинга (длина) и максимальное число символов в строке листинга (ширина). Задаваемая длина должна находиться в пределах от 10 до 255 (значение по умолчанию — 50). Ширина может варьироваться от 60 до 132 (значение по умолчанию — 80). Если длина не указана, ширине, если она задана, должна предшествовать запятая.

Вторая форма директивы **PAGE** (со знаком +) означает, что номер секции программы должен быть увеличен, а номер страницы устанавливается равным 1. Номер страницы листинга состоит их 2-х компонент: номера секции и номера страницы внутри секции и имеет следующий вид:

```
секция-страница
```

По умолчанию нумерация страниц листинга начинается со значения 1-1.

Третья форма директивы **PAGE** (без аргументов) предписывает MASM перейти на новую страницу листинга.

Директива **.XLIST** подавляет копирование в листинг последующих строк исходной программы. Следует помнить, что обработка MASM директивы **.XLIST** перекрывает все предшествующие ей директивы управления листингом.

Директива **.LIST** восстанавливает копирование в листинг последующих строк исходной программы.

Директива **.LALL** сообщает MASM, что листинг должен содержать все предложения макрорасширений, если они есть, включая обычные комментарии (с предшествующей **;**), но исключая макрокомментарии (с предшествующей **;;**).

По директиве **.XALL** распечатываются только предложения макрорасширений, по которым генерируется программный код или данные. Комментарии игнорируются. Этот режим действует по умолчанию при отсутствии в исходной программе директив **.LALL** и **.SALL**.

Директива **.SALL** подавляет распечатку макрорасширений.

Директива **.SFCOND** подавляет распечатку тел всех последующих условных блоков, условия ассемблирования которых окажутся ложными.

Директива **.LFCOND** восстанавливает распечатку таких блоков.

Директива **.TFCOND** устанавливает умалчиваемый режим распечатки условных блоков. Эта директива работает в сочетании с опцией **/X** MASM. Если опция **/X** при запуске MASM не была задана, **.TFCOND** разрешает печать условных блоков с ложными условиями. Если же опция **/X** была задана, **.TFCOND** подавляет печать таких блоков. Каждая обработка Ассемблером директивы **.TFCOND** меняет режим распечатки условных блоков с ложными условиями на противоположный.

Директива **.XCREF** подавляет генерацию листинга перекрестных ссылок для меток, переменных и имен. Если в директиве задан список имен, из листинга перекрестных ссылок исключаются только указанные объекты, а все остальные в листинг попадают. Если список содержит более, чем одно имя, его элементы должны отделяться друг от друга запятыми.

Директива **.CREF** восстанавливает генерацию листинга перекрестных ссылок.



Другие директивы

В языке Ассемблере имеются еще 2 директивы, имеющие некоторое отношение к вводу/выводу. Это директивы **%OUT** и **NAME**.

Директива **%OUT** имеет следующий формат:

```
%OUT текст
```

При обработке Ассемблером этой директивы указанный текст выдается на системный экран. Эта директива полезна при отслеживании прохождения процесса ассемблирования через специфичные участки очень большой исходной программы.

Директива **%OUT** срабатывает на обоих проходах MASM. Для того, чтобы сообщение выдавалось на каком-либо одном проходе, можно использовать директивы **IF1** и **IF2**.

Пример:

```
IF1
%OUT First pass
ENDIF
```

Сообщение будет выдано только на 1-м проходе.

Директива **NAME** позволяет присваивать имя текущему модулю и имеет формат:

```
NAME имя-модуля
```

Имя модуля используется программой **LINK** при выдаче диагностических сообщений. **Имя-модуля** может быть любой комбинацией букв и цифр, из которой используются только первые 6 символов. Имя должно быть уникальным и не может совпадать с ключевым словом.

При отсутствии в исходной программе директивы **NAME** Ассемблер создает умалчиваемое имя модуля, в качестве которого используются первые 6 символов текста, задаваемого директивой **TITLE**. Если же и директива **TITLE** опущена, модулю присваивается имя **A**.



Глобальные объявления

Директивы глобального объявления позволяют определять метки, переменные и абсолютные имена, доступ к которым возможен из всех сегментов программы независимо от того, как они ассемблировались.

В языке Ассемблера имеются 2 директивы глобального объявления: **PUBLIC** и **EXTRN**, которые дополняют друг друга.

Директива **PUBLIC** имеет следующий формат:

```
PUBLIC имя,...
```

Директива **PUBLIC** делает указанные в списке переменные, метки или абсолютные имена, общими, то есть, доступными всем модулям программы. Каждый элемент списка должен быть определен в текущем исходном файле. Абсолютные имена, если они указаны, должны представлять одно- или двухбайтные целые числа или строковые значения.

Обычно до копирования в объектный файл строчные буквы в общих именах преобразуются в заглавные. Для сохранения первоначального написания имен с учетом регистра могут быть использованы опции **/ML** и **/MX** MASM.

Директивой **PUBLIC** должны быть описаны имена, используемые при работе с **SYMDEB** в режиме символьной отладки.

Директива **EXTRN** имеет следующий формат:

```
EXTRN имя:тип,...
```

Директива **EXTRN** определяет внешние по отношению к текущему сегменту объекты с указанием их типа. Внешним объектом может быть переменная, метка или имя, объявленные в другом модуле программы директивой **PUBLIC**. Тип, указанный в директиве **EXTRN**, должен соответствовать действительному типу объекта и может кодироваться одним из следующих ключевых слов:

- **BYTE**
- **WORD**
- **DWORD**
- **QWORD**

- **TBYTE**
- **NEAR**
- **FAR**
- **ABS**.

Тип **ABS** используется при описании имен, представляющих абсолютные значения. Остальные описатели типа имеют обычный смысл.

Несмотря на то, что действительные адреса объектов до обработки объектного файла программой **LINK** неизвестны, Ассемблер, основываясь на местонахождении директивы **EXTRN**, может делать предположения о том, какой сегмент следует использовать для адресации внешнего объекта. Если директива **EXTRN** находится внутри какого-либо сегмента, считается, что указанные внешние объекты связаны с этим сегментом, и описывающие их директивы **PUBLIC** должны содержаться в сегменте с тем же именем и теми же атрибутами.

Если директива **EXTRN** не входит ни в один сегмент, предположений о содержащем внешний объект сегменте не делается, и соответствующая директива **PUBLIC** может располагаться в любом сегменте. В обоих случаях для подавления адресации внешнего объекта относительно умалчиваемого сегмента может использоваться оператор переключения сегмента (**:**).



Инструкции процессоров

Набор инструкций процессора представляет собой самый нижний (машинный) уровень программного обеспечения компьютера. Программирование на этом уровне представляет из себя весьма непростую задачу по той причине, что выполняемые каждой инструкцией функции довольно просты, и поэтому любая сколько-нибудь сложная программа включает в себя большое количество инструкций и обычно теряет наглядность. Кроме того, программист обязан знать многие архитектурные и функциональные особенности компьютера, о которых он может не думать при программировании на языках высокого уровня. С другой стороны, только набор машинных инструкций предоставляет программисту все возможности компьютера.

Инструкции процессора доступны пользователю через программу MASM.

Инструкции процессора в унифицированной форме обрабатывают различные типы операндов. Почти каждая инструкция может оперировать с байтом или словом на выбор. Регистры, переменные памяти (поля памяти, представленные адресами) и непосредственные операнды могут кодироваться в большинстве инструкций без ограничений, за исключением, разумеется, того, что непосредственный операнд может быть только источником, но не приемником. В частности, переменные памяти могут участвовать в операциях сложения, вычитания, сдвига, сравнения и других без предварительного помещения их в регистры, что экономит инструкции, регистры и время выполнения ассемблерной программы. Для языков высокого уровня, где большинство переменных размещено в памяти, компиляторы могут генерировать более быстродействующий и меньший по объему объектный код.

Набор инструкций процессоров можно рассматривать как состоящий из 2-х уровней:

- ассемблерный уровень;
- машинный уровень.

Ассемблерный уровень включает в себя около 100 инструкций. Например, только инструкция **MOV** способна пересылать содержимое регистра или ячейки памяти или непосредственное значение в регистр или ячейку памяти. Процессор распознает 28 различных инструкций для **MOV** (пересылка байта из регистра в память, пересылка непосредственно представленного слова в регистр).

Ассемблер транслирует написанные программистом инструкции ассемблерного уровня в инструкции машинного уровня, которые непосредственно выполняются процессором. Компиляторы языков высокого уровня транслируют предложения своего языка также в инструкции машинного уровня.

Наличие 2-х уровней инструкций направлено на удовлетворение 2-х различных требований: эффективности программы и относительной простоты программирования. Набор инструкций машинного уровня (их около 300) позволяет эффективно использовать память. Например, машинная инструкция, предназначенная для увеличения на 1 операнда памяти, занимает 3-4 байта, так как она должна содержать адрес операнда. Для наращивания регистра эта информация не нужна, и инструкция может быть короче. Если программист будет использовать одну инструкцию для наращивания регистра, другую — для операнда памяти, выгода от компактных инструкций будет сведена на нет сложностью программирования. Инструкции ассемблерного уровня с точки зрения программиста проще. Программист кодирует одну форму инструкции **INC**, а Ассемблер проверяет ее операнд и решает, какую генерировать машинную инструкцию.



Инструкции пересылки данных

14 инструкций этого типа обеспечивают пересылку одиночных байтов и слов между памятью и регистрами, а также между портами ввода/вывода и регистрами **AL** или **AX**. В эту группу включены также инструкции манипуляции со стеком и инструкции пересылки флагов процессора и загрузки регистров сегмента.

Все инструкции пересылки данных можно условно разделить на 4 группы:

Общего назначения

- **MOV** — пересылка байта или слова;
- **PUSH** — сохранение слова в стеке;
- **POP** — восстановление слова из стека;
- **XCLHG** — обмен байтами или словами;
- **XLAT** — трансляция байта.

Ввода/вывода

- **IN** — ввод байта или слова из порта;
- **OUT** — вывод байта или слова в порт.

Адресные операции

- **LEA** — загрузка исполнительного адреса;
- **LDS** — загрузка указателя с использованием **DS**;
- **LES** — загрузка указателя с использованием **ES**.

Пересылка флагов

- **LAHF** — загрузка флагов в **AH**;
- **SAHF** — установка флагов из **AH**;
- **PUSHF** — сохранение флагов в стеке;
- **POPF** — восстановление флагов из стека.

Все инструкции пересылки данных, кроме **POPF** и **SAHF**, значений флагов процессора не изменяют.



Инструкции общего назначения

MOV приемник,источник

Пересылка байта или слова. Байт или слово пересылается из источника в приемник.

PUSH источник

Сохранение слова в стеке. Указатель стека (регистр **SP**) уменьшается на 2, и в вершину стека помещается слово из источника. Часто **PUSH** используется для занесения в стек параметров процедуры перед ее вызовом. В общем случае это основное средство для сохранения временных данных.

POP приемник

Восстановление слова из стека. Слово данных из текущей вершины стека, адресуемой регистром **SP**, пересылается в операнд приемник. Регистр **SP** затем увеличивается на 2 и указывает на новую вершину стека. **POP** может использоваться для восстановления из стека временных данных.

XCLHG приемник,источник

Обмен байтами или словами. Эта инструкция осуществляет обмен содержимым (байт или слово) между операндами источник и приемник.

При использовании в сочетании с префиксом **LOCK XCLHG** может проверять и устанавливать семафор, управляющий доступом к разделяемому несколькими процессорами ресурсу.

XLAT таблица-трансляции

Трансляция байта. Байт в регистре **AL** замещается байтом из созданной пользователем 256-байтной таблицы трансляции. Предполагается, что регистр **BX** содержит адрес начала этой таблицы. Содержимое **AL** используется как индекс в таблице и замещается байтом, выбираемом из таблицы со смещением, соответствующим двоичному содержимому регистра **AL** (первый байт таблицы имеет смещение 0). Инструкция **XLAT** может использоваться для трансляции символов из одного кода в другой.



Ввод/вывод

IN аккумулятор,порт

Ввод байта или слова из порта. Байт или слово, полученные из указанного порта, помещается в регистр **AL** или **AX** соответственно. Номер порта может задаваться либо непосредственно числом в пределах от 0 до 255, обеспечивающим доступ только к этим портам, либо указанием регистра **DX**, который предварительно должен быть загружен требуемым значением, что обеспечивает доступ к портам с номерами от 0 до 65535.

OUT порт,аккумулятор

Вывод байта или слова в порт. Байт или слово, предварительно помещенные в регистр **AL** или **AX** соответственно, выводится в указанный порт. Номер порта может задаваться либо непосредственно числом в пределах от 0 до 255, обеспечивающим доступ только к этим портам, либо указанием регистра **DX**, который предварительно должен быть загружен требуемым значением, что обеспечивает доступ к портам с номерами от 0 до 65535.



Адресные операции

Адресные операции работают не с содержимым или значением переменных, а с их адресами. Они наиболее полезны при обработке списков, базированных переменных и в операциях со строками.

LEA приемник,источник

Загрузка исполнительного адреса. Смещение операнда источник помещается в операнд приемник. Источник должен быть операндом памяти, а приемник — 16-битовым регистром. Эта инструкция может использоваться для установки регистров перед **XLAT** или операциями со строками, которые предполагают, что некоторые регистры загружены адресными значениями.

LDS приемник,источник

Загрузка указателя с **DS**. Указатель представляет собой 32-битовую адресную переменную, первое слово которой содержит смещение, а второе — базовый адрес (сегмент). Адрес указателя в этой инструкции задается операндом источник, который должен быть операндом памяти. Слово смещения из указателя помещается в операнд приемник, в качестве которого может быть указан 16-битовый регистр. Слово сегмента из указателя помещается в регистр **DS**. Указание **SI** в качестве приемника является обычным способом подготовки для строковой операции строки-источника, расположенной вне текущего сегмента данных.

LES приемник,источник

Загрузка указателя с **ES**. Указатель представляет собой 32-битовую адресную переменную, первое слово которой содержит смещение, а второе — базовый адрес (сегмент). Адрес указателя в этой инструкции задается операндом источник, который должен быть операндом памяти. Слово смещения из указателя помещается в операнд приемник, в качестве которого может быть указан 16-битовый регистр. Слово сегмента из указателя помещается в регистр **ES**. Указание **DI** в качестве приемника является обычным способом подготовки для строковой операции строки-приемника, расположенной вне текущего экстра сегмента.



Операции с флагами

LAHF

Загрузка флагов в **AH**. Флаги **SF**, **ZF**, **AF**, **PF** и **CF** копируются в биты 7, 6, 4, 2 и 0 соответственно регистра **AH**. Биты 5, 3 и 1 не определены. Сами флаги не изменяются.

SAHF

Установка флагов из **AH**. Биты 7, 6, 4, 2 и 0 регистра **AH** замещают значения флагов **SF**, **ZF**, **AF**, **PF** и **CF** соответственно. Значения флагов **OF**, **DF**, **IF** и **TF** не изменяются.

PUSHF

Сохранение флагов в стеке. По этой инструкции указатель стека **SP** уменьшается на 2, и в вершину стека помещаются все флаги процессора в формате слова согласно их расположению в регистре флагов.

Сами флаги не изменяются.

POPF

Восстановление флагов из стека. Из вершины стека, адресуемой регистром **SP**, в регистр флагов процессора помещаются специфичные биты, соответствующие расположению флагов в регистре флагов.

После этого **SP** увеличивается на 2 и указывает на новую вершину стека.

Комбинации инструкций **PUSHF** и **POPF** позволяют процедуре сохранять и восстанавливать флаги вызвавшей ее программы. Кроме того, таким образом можно устанавливать значение флага **TF** (ловушка), так как специальной инструкции для этого нет.

Для этого следует сохранить флаги в стеке, изменить значение бита 8 и затем восстановить флаги из стека.



Арифметические инструкции

Форматы арифметических данных

Арифметические операции могут выполняться над операндами 4-х типов:

- Двоичные без знака.
- Двоичные со знаком (целые).
- Упакованные десятичные без знака.
- Распакованные десятичные без знака.

Двоичные числа могут занимать 1 или 2 байта. Десятичные числа хранятся побайтно по 2 десятичной цифре на байт для упакованного формата или по 1 десятичной цифре на байт для распакованного формата. Процессор предполагает, что определенные в арифметических инструкциях операнды содержат данные, представляющие корректные для данной инструкции числа. Некорректные данные могут привести к непредсказуемым результатам.

Двоичные числа без знака могут занимать 8 или 16 бит; все биты значимы. Диапазон значений 8-битового числа — от 0 до 255, 16-битово го — от 0 до 65535. Над двоичными числами без знака можно выполнять операции сложения, вычитания, умножения и деления.

Двоичные числа со знаком (целые) могут занимать также 8 или 16 бит. Значение старшего бита (самого левого) задает знак числа: 0 — положительное, 1 — отрицательное. Отрицательные числа представляются стандартным дополнением до 2. Поскольку один разряд отведен под знак, диапазон изменения 8-битового числа — от -127 до +127, 16-битового — от -32768 до +32767. Число 0 имеет положительный знак. Над двоичными числами со знаком могут быть выполнены операции умножения и деления. Сложение и вычитание выполняются без учета знака. Для обнаружения переноса в знаковый разряд в результате беззнаковой операции можно использовать инструкции условного перехода или **INTO**.

Упакованные десятичные числа хранятся как беззнаковые байтовые величины. Каждый байт содержит 2 десятичные цифры, занимающие по 4 бита каждая. Цифра в старшем полубайте более значима. В каждом полубайте допустимы только 16-ричные значения от 0 до 9; соответственно пределы изменения десятичного числа — от 0 до 99. Сложение и вычитание таких чисел выполняются в 2 стадии. Сначала применяется обычная беззнаковая двоичная инструкция, которая формирует в регистре **AL** промежуточный результат. Затем выполняется операция настройки (инструкция **DAA** или **DAS**), преобразующая содержимое **AL** в корректный упакованный десятичный результат. Умножение и деление упакованных десятичных чисел невозможно.

Распакованные десятичные числа хранятся как беззнаковые байтовые величины. Десятичная цифра располагается в младшем полубайте. Допустимы и интерпретируются как десятичные числа 16-ричные значения от 0 до 9. Для выполнения операций умножения и деления старший полубайт должен быть заполнен нулями; для сложения и вычитания он может содержать любое значение.

Арифметические операции над распакованными десятичными числами выполняются в две стадии. Сначала используются обычные беззнаковые инструкции сложения, вычитания или умножения, которые формируют в регистре **AL** промежуточный результат. Затем выполняется операция настройки (инструкция **AAA**, **AAS** или **AAM**), преобразующая содержимое **AL** в результирующее корректное распакованное десятичное число. Деление выполняется аналогично, за исключением того, что сначала следует настроить числитель в **AL** (инструкция **AAD**), а затем выполнить инструкцию беззнакового двоичного деления, результатом которого будет корректное распакованное десятичное число.

Формат десятичных распакованных чисел подобен представлению десятичных цифр в коде ASCII. При этом для числа в коде ASCII старший полубайт содержит 16-ричное значение 3. Возможное содержимое старшего полубайта для распакованного формата приведено выше. Преобразование из одного вида в другой сложности не представляет.



Арифметические операции и флаги

Арифметические инструкции оставляют после своего выполнения некоторые характеристики результатов операций в виде значений 6 флагов. Большинство из них могут анализироваться последующими инст-

рукциями условного перехода; может также использоваться инструкция прерывания по переполнению **INTO**. Влияние каждой инструкции на флаги указано при описании инструкции.

Однако имеются следующие общие правила:

1. Флаг переноса **CF** устанавливается в 1, если в результате операции сложения был перенос из старшего бита или в результате операции вычитания был заем в старший бит результата. Если же переноса или заема не было, **CF** устанавливается в 0. Заметим, что знаковый перенос характеризуется различными значениями флагов **CF** и **OF**. Флаг **CF** может использоваться для обнаружения беззнакового переполнения. Следует помнить, что две инструкции, **ADC** (сложение с переносом) и **SBB** (вычитание с заемом) вовлекают **CF** в свои операции и могут быть поэтому использованы для мультибайтного (32-, 64-разрядного) сложения и вычитания.

2. Флаг промежуточного переноса **AF** устанавливается в 1 при переносе из младшего полубайта результата во время сложения или при заеме в младший полубайт результата во время вычитания. Если же переноса или заема не было, **AF** устанавливается в 0. Флаг **AF** введен для выполнения десятичной настройки и обычно в других целях не используется.

3. Флаг знака **SF** устанавливается арифметическими и логическими инструкциями равным старшему (7-му или 15-му) биту результата. Для двоичных чисел со знаком **SF** будет равен 0 в случае положительного результата и 1 — в случае отрицательного (если нет переполнения). Значение флага **SF** может анализироваться после сложения или вычитания инструкциями условного перехода. Программы, выполняющие беззнаковые операции, обычно игнорируют **SF**, так как старший бит результата в этом случае интерпретируется как двоичная цифра, а не как знак.

4. Флаг нуля **ZF** устанавливается в 1, если результат арифметической или логической операции равен 0, и устанавливается в 0, если результат отличен от 0. Значение флага может анализироваться инструкциями условного перехода.

5. Флаг паритета **PF** устанавливается в 1, если младшие 8 бит результата арифметической или логической операции содержат четное число единиц, и устанавливается в 0, если число единиц нечетно. Флаг **PF** может использоваться для контроля символов в коде ASCII на корректность паритета.

6. Флаг переполнения **OF** устанавливается в 1, если результат слишком велик для положительного числа или слишком мал для отрицательного и не помещается в операнд-приемник (не считая знаковый разряд). В противном случа значение OF — 0. Состояние этого флага отражает наличие арифметического переполнения со знаком. Он может анализироваться инструкциями условного перехода или инструкцией **INTO**. В беззнаковых операциях **OF** обычно игнорируется.



Сложение

ADD приемник,источник

Сложение байтов или слов. Арифметическая сумма 2-х операндов, которыми могут быть байты или слова, замещает операнд-приемник. Оба операнда могут содержать двоичные числа со знаком или без него. **ADD** модифицирует флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**.

ADC приемник,источник

Сложение с переносом. Эта инструкция выполняет арифметическую сумму своих операндов, добавляет 1, если установлен в 1 флаг **CF**, и помещает результат на место операнда-приемника. Оба операнда могут содержать двоичные числа со знаком или без него. **ADC** модифицирует флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**. Поскольку **ADC** использует перенос от предыдущей операции, она может применяться для сложения чисел длиннее 2 байтов.

INC приемник

Увеличение байта или слова на 1.

К содержимому приемника прибавляется 1. Операнд может быть байтом или словом и рассматривается как двоичное число без знака. **INC** модифицирует флаги **AF**, **OF**, **PF**, **SF** и **ZF** и не влияет на **CF**.

AAA

ASCII-настройка для сложения. Эта инструкция преобразует содержимое регистра **AL** в корректное распакованное десятичное число; старший полубайт обнуляется. **AAA** модифицирует флаги **AF** и **CF**; состояния флагов **OF**, **PF**, **SF** и **ZF** после **AAA** не определены.

DAA

Десятичная настройка для сложения. Эта инструкция корректирует результат предшествующего сложения 2-х правильных упакованных десятичных чисел, содержащихся в регистре **AL**. Содержимое **AL** преобразуется в пару корректных упакованных десятичных чисел. **DAA** модифицирует флаги **AF**, **CF**, **PF**, **SF** и **ZF**; состояние флага **OF** после **DAA** не определено.



Вычитание

SUB приемник,источник

Вычитание байтов или слов. Содержимое источника вычитается из содержимого приемника, и результат помещается на место приемника. Операнды могут быть байтами или словами и содержать двоичные числа со знаком или без него. **SUB** модифицирует флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**.

SBB приемник,источник

Вычитание с заемом. По этой инструкции содержимое источника вычитается из содержимого приемника, из результата вычитается еще 1, если установлен в 1 флаг **CF**, и окончательный результат помещается на место приемника. Оба операнда могут быть байтами или словами и содержать двоичные числа со знаком или без него. **SBB** модифицирует флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**. Поскольку **SBB** использует заем предыдущей операции, она может применяться для вычитания чисел длиннее 2 байтов.

DEC приемник

Уменьшение байта или слова на 1. Содержимое приемника уменьшается на 1. Приемник может быть байтом или словом. **DEC** модифицирует флаги **AF**, **OF**, **PF**, **SF** и **ZF** и не влияет на состояние **CF**.

NEG приемник

Отрицание байта или слова. По этой инструкции содержимое приемника, который может быть байтом или словом, вычитается из 0, и результат помещается на место приемника. Эта форма дополнения до 2 используется для изменения знака целого числа. Если операнд равен 0, его знак не изменяется.

Попытка отрицания байта, содержащего -128, или слова, содержащего -32768, не изменяет операнд, но устанавливает в 1 флаг **OF**. **NEG** модифицирует флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**. Флаг **CF** всегда устанавливается в 1, исключая тот случай, когда операнд равен 0 (тогда и **CF=0**).

CMP приемник,источник

Сравнение байтов или слов. Инструкция **CMP** вычитает содержимое источника из содержимого приемника, но результат не возвращает.

Операнды, которые могут быть байтами или словами, не изменяются, но модифицируются флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF**, что может анализироваться последующими инструкциями условного перехода.

Состояния этих флагов отражают отношение приемника к источнику. Например, если после **CMP** следует инструкция **JG**, переход произойдет, если содержимое приемника больше содержимого источника.

AAS

ASCII-настройка для вычитания. Эта инструкция преобразует находящийся в регистре **AL** результат предшествующей операции вычитания 2-х корректных десятичных распакованных чисел в корректное десятичное распакованное число, остающееся также в **AL**.

Старший полубайт регистра **AL** обнуляется. **AAS** модифицирует флаги **AF** и **CF**; состояния флагов **OF**, **PF** и **ZF** после **AAS** не определены.

DAS

Десятичная настройка для вычитания.

Эта инструкция преобразует находящийся в регистре **AL** результат предшествующей операции вычитания 2-х корректных десятичных упакованных чисел в пару корректных десятичных упакованных цифр, остающихся также в **AL**. **DAS** модифицирует флаги **AF**, **CF**, **PF**, **SF** и **ZF**; состояние флага **OF** не определено.



Умножение

MUL источник

Умножение байтов или слов без знака. Инструкция **MUL** выполняет беззнаковое умножение содержимых источника и аккумулятора. Если источник является байтом, в качестве аккумулятора используется регистр **AL**, а результат двойной длины помещается в регистры **AH** и **AL**. Если источник является словом, в качестве аккумулятора используется регистр **AX**, а результат двойной длины помещается в регистры **DX** и **AX**. Операнды рассматриваются как двоичные числа без знака. Если старшая половина результата (содержимое **AH** для источника-байта или содержимое **DX** для источника-слова) не равна 0, флаги **CF** и **OF** устанавливаются в 1, в противном случае — в 0. Когда **CF** и **OF** установлены в 1, это означает, что **AH** или **DX** содержит значимые цифры результата. Состояния флагов **AF**, **PF**, **SF** и **ZF** после **MUL** не определены.

IMUL источник

Целочисленное умножение байтов или слов. Инструкция **IMUL** выполняет умножение со знаком содержимых источника и аккумулятора. Если источник является байтом, в качестве аккумулятора используется регистр **AL**, а результат двойной длины помещается в регистры **AH** и **AL**. Если источник является словом, в качестве аккумулятора используется регистр **AX**, а результат двойной длины помещается в регистры **DX** и **AX**. Если старшая половина результата (содержимое **AH** для источника-байта или содержимое **DX** для источника-слова) не является расширением знака младшей половины, флаги **CF** и **OF** устанавливаются в 1, в противном случае — в 0. Когда **CF** и **OF** установлены в 1, это означает, что **AH** или **DX** содержит значимые цифры результата. Состояния флагов **AF**, **PF**, **SF** и **ZF** после **IMUL** не определены.

AAM

ASCII-настройка для умножения.

Инструкция **AAM** корректирует результат предшествующей операции умножения 2-х корректных десятичных распакованных операндов. Корректное десятичное распакованное число, состоящее из 2-х цифр, извлекается из регистров **AH** и **AL**, и результат возвращается в регистры **AH** и **AL**. Старшие полубайты перемножаемых операндов должны быть обнулены, что необходимо **AAM** для формирования правильного результата. **AAM** модифицирует флаги **PF**, **SF** и **ZF**; состояния флагов **AF**, **CF** и **OF** после **AAM** не определены.



Деление

DIV источник

Деление байтов или слов без знака. Инструкция **DIV** выполняет беззнаковое деление содержимого аккумулятора (и его расширения) на содержимое источника. Если источник является байтом, предполагается, что делимое расположено в регистрах **AH** и **AL**. Частное остается в регистре **AL**, остаток — в регистре **AH**. Если источник является словом, предполагается, что делимое расположено в регистрах **DX** и **AX**. Частное в этом случае остается в регистре **AX**, остаток — в регистре **DX**. Если значение частного превосходит вместимость регистра-приемника (FFh для источника-байта или FFFFh для источника-слова), возникает ситуация «деление на 0» и генерируется прерывание с номером 0; частное и остаток в этом случае не определены.

Дробная часть частного отсекается. После **DIV** состояния флагов **AF**, **CF**, **OF**, **PF**, **SF** и **ZF** не определены.

IDIV источник

Целочисленное деление байтов или слов. Инструкция **IDIV** выполняет деление со знаком содержимого аккумулятора (и его расширения) на содержимое источника. Если источник является байтом, предполагается, что делимое расположено в регистрах **AH** и **AL**. Частное остается в регистре **AL**, остаток — в регистре **AH**. Для такого деления максимально допустимое положительное частное равно 127 (7Fh), а минимально допустимое отрицательное частное равно -127 (81h). Если источник является словом, предполагается, что делимое расположено в регистрах **DX** и **AX**. Частное в этом случае остается в регистре **AX**, остаток — в регистре **DX**. Для такого деления значение частного может находиться в пределах от -32767 (8001h) до 32767 (7FFFh). Если частное положительно и превышает положительный максимум или отрицательно и меньше отрицательного минимума, генерируется прерывание с номером 0 (деле-

ние на 0); частное и остаток в этом случае не определены. Нецелочисленное частное округляется до целого числа (по направлению к 0). Остаток имеет тот же знак, что и делимое. После **IDIV** состояния флагов **AF**, **CF**, **OF**, **ZF**, **PF** и **SF** не определены.

AAD

ASCII-настройка для деления. Инструкция **AAD** модифицирует числитель в регистре **AL** перед делением 2-х корректных десятичных распакованных операндов таким образом, чтобы частное от деления было также корректным десятичным распакованным числом. Для того, чтобы последующая инструкция **DIV** сформировала правильный результат, регистр **AH** должен содержать нули. Частное остается в регистре **AL**, остаток — в регистре **AH**; оба старших полубайта обнуляются. **AAD** модифицирует флаги **PF**, **SF** и **ZF**; состояния флагов **AF**, **CF** и **OF** после **AAD** не определены.

CBW

Преобразование байта в слово. Инструкция **CBW** заполняет регистр **AH** битами, равными знаковому биту однобайтного числа в регистре **AL**. **CBW** на состояния флагов не влияет. Эта инструкция может использоваться для получения делимого двойного размера (слова) из результата предшествующей операции деления байтов.

CWD

Преобразование слова в двойное слово. Инструкция **CWD** заполняет регистр **DX** битами, равными знаковому биту двухбайтного числа в регистре **AX**. **CWD** на состояния флагов не влияет. Эта инструкция может использоваться для получения делимого двойного размера (двойного слова) из результата предшествующей операции деления слов.



Инструкции обработки бит

Имеется 3 группы инструкций для обработки бит в форматах байта или слова: логические, сдвиги и вращения (все операции выполняются и над байтами, и над словами).

Логические инструкции

Логические инструкции включают булевы операторы НЕ, И, включающие ИЛИ, исключающие ИЛИ и операцию тестирования, которая устанавливает флаги, но не изменяет значения своих операндов.

Инструкции **AND**, **OR**, **XOR** и **TEST** следующим образом влияют на флаги. Флаги **OF** и **CF** логическими инструкциями всегда устанавливается в 0, а состояние флага **AF** не определено. Состояния флагов **SF**, **ZF** и **PF** отражают результат операции и могут анализироваться инструкциями условного перехода. Интерпретация этих флагов такая же, как для арифметических инструкций. Флаг **SF** устанавливается в 1, если результат отрицателен (старший бит равен 1), и устанавливается в 0, если результат положителен (старший бит равен 0). Флаг **ZF** устанавливается в 1, если результат равен 0, и устанавливается в 0 в противном случае. Флаг **PF** устанавливается в 1, если результат содержит четное число единиц, и устанавливается в 0 в противном случае. Инструкция **NOT** на состояния флагов не влияет.

NOT приемник

Отрицание. Инструкция **NOT** инвертирует все биты (формирует дополнение до 1) байта или слова.

AND приемник,источник

Логическое И. Инструкция **AND** выполняет операцию логическое И двух операндов (байтов или слов) и возвращает результат в операнд-приемник. Бит результата устанавливается в 1, если установлены в 1 оба соответствующих ему бита операндов, и устанавливается в 0 в противном случае.

OR приемник,источник

Включающее ИЛИ. Инструкция **OR** выполняет операцию логическое включающее ИЛИ двух операндов (байтов или слов) и помещает результат на место операнда-приемника. Бит результата устанавливается в 1, если равен 1 хотя бы один из 2-х соответствующих ему битов операндов, и устанавливается в 0 в противном случае.

XOR приемник,источник

Исключающее ИЛИ. Инструкция **XOR** выполняет операцию логическое исключающее ИЛИ 2-х операндов и помещает результат на место операнда-приемника. Бит результата устанавливается в 1, если соответствующие ему биты операндов имеют противоположные значения, и устанавливается в 0 в противном случае.

TEST приемник,источник

Тестирование. Инструкция **TEST** выполняет операцию логическое И двух операндов (байтов или слов), модифицирует флаги, но результат не возвращает, то есть,операнды не изменяются. Если за **TEST** следует инструкция **JNZ** (переход, если не 0), то переход будет иметь место, если в обоих операндах имеются единицы в совпадающих позициях.



Сдвиги

Биты в байтах или словах могут быть сдвинуты арифметически или логически. В соответствии с кодируемым в инструкции счетчиком может быть выполнено до 255 сдвигов. Счетчик может быть специфицирован как константа 1 или как регистр **CL**, что позволяет задавать величину сдвига в процессе работы программы. Арифметические сдвиги могут быть использованы для умножения и деления двоичных чисел на степени 2. Логические сдвиги могут применяться для выделения битов в байтах или словах.

Инструкции сдвига следующим образом воздействуют на флаги. Состояние флага **AF** всегда не определено после операции сдвига. Воздействие на флаги **PF**, **SF** и **ZF** аналогично логическим инструкциям. Флаг **CF** всегда содержит значение последнего сдвинутого за пределы операнда приемник бита. Состояние флага **OF** после многобитного сдвига всегда не определено. При единичном сдвиге **OF** устанавливается в 1, если в результате операции знаковый бит изменил свое значение, и устанавливается в 0 в противном случае.

SHL/SAL приемник,счетчик

Сдвиг влево. Инструкции **SHL** и **SAL** выполняют операции соответственно логического и арифметического сдвига влево операнда приемник на величину бит, определяемую счетчиком. Приемник может быть байтом или словом. Появляющиеся справа биты заполняются нулями. Если знаковый бит сохраняет первоначальное значение, флаг **OF** устанавливается в 0.

SHR приемник,источник

Логический сдвиг вправо. Инструкция **SHR** сдвигает биты операнда приемник (байта или слова) вправо на число разрядов, определяемое операндом счетчик. Появляющиеся слева биты заполняются нулями. Если знаковый бит сохраняет свое первоначальное значение, флаг OF устанавливается в 0.

SAR приемник,счетчик

Арифметический сдвиг вправо. Инструкция **SAR** сдвигает биты операнда приемник (байта или слова) вправо на число разрядов, определяемое операндом счетчик. Биты, равные первоначальному знаковому биту, появляются слева, сохраняя таким образом первоначальный знак числа. Отметим, что результат выполнения **SAR** отличается от делимого «эквивалентной» операции **IDIV**, если операнд приемника отрицателен и за его пределы сдвигаются единицы. Например, сдвиг числа -5 вправо на 1 бит дает -3, а деление -5 на 2 дает -2. Различие инструкций заключается в том, что **IDIV** округляет все числа по направлению к 0, а SAR округляет положительные числа к 0 и отрицательные — от нуля.



Вращения

Биты в байтах и словах можно вращать. Биты, сдвигаемые за пределы операнда, не теряются, как при сдвиге, а циклически появляются с другой стороны операнда. Как при сдвиге, величина вращения задается операндом счетчик, который может быть специфицирован как константа 1 или как регистр **CL**. Флаг **CF** может выступать как расширение операнда в двух инструкциях вращения (RCL и RCR), позволяя выделять бит во флаг **CF** и затем проверить его значение инструкциями **JC** или **JNC**. Вращения воздействуют только на флаг переноса **CF** и флаг переполнения **OF**.

Флаг **CF** всегда содержит значение последнего вышедшего за операнд бита. В многопозиционных вращениях состояние флага **OF** всегда не определено. В одиночном вращении **OF** устанавливается в 1, если операция изменяет значение старшего (знакового) бита операнда, и устанавливается в 0 в противном случае.

ROL приемник,счетчик

Вращение влево. Инструкция **ROL** вращает байт или слово приемника влево на число бит, определяемое счетчиком.

ROR приемник,счетчик

Вращение вправо. Инструкция **ROR** работает аналогично **ROL**, но вправо.

RCL приемник,счетчик

Вращение влево с переносом. Инструкция **RCL** вращает биты байта или слова приемника влево на число бит, определяемое счетчиком. Флаг **CF** рассматривается как часть приемника, то есть, его значение при этом вращении попадает в младший бит приемника, а сам **CF** принимает значение старшего бита приемника.

RCR приемник,счетчик

Вращение вправо с переносом. Инструкция **RCR** работает в точности как **RCL** с той лишь разницей, что биты вращаются вправо.



Инструкции обработки строк

Пять базовых строковых операций, называемых примитивами, позволяют оперировать со строками байтов или слов по одному элементу (байту или слову) за раз. Эти операции могут обрабатывать строки длиной до 64K. Операции со строками обеспечивают пересылку, сравнение, сканирование строк по значению, а также пересылку элементов строки в аккумулятор или из него. Этим базовыми инструкциям может предшествовать однобайтный префикс, наличие которого обеспечивает многократное повторение инструкции аппаратным способом, что гарантирует более высокое быстродействие, чем в случае программного цикла. Процесс повторения может быть прекращен при возникновении различных ситуаций, а сама повторяемая операция может быть как прервана, так и возобновлена.

Строковые инструкции во многом похожи друг на друга. Они могут иметь операнд-приемник, операнд-источник или оба эти операнда.

Аппаратно предполагается, что исходная строка размещена в текущем сегменте данных (для ее адресации используется регистр **DS**); для изменения этого допущения может использоваться однобайтный префикс изменения сегмента. Строка-приемник должна размещаться в текущем экстра сегменте (для ее адресации используется регистр **ES**). Для проверки того, что является элементом строки (байт или слово), Ассемблер проверяет атрибуты операндов инструкции. Однако, в действительности эти операнды для адресации строк не используются. Для адресации используются регистры **SI**, который предполагается загруженным значением смещения строки-источника относительно содержимого **DS**, и **DI**, содержимое которого трактуется как смещение строки-приемника относительно содержимого **ES**. Все эти регистры должны быть загружены требуемыми значениями до выполнения строковой операции, для чего могут использоваться инструкции **LDS**, **LES** и **LEA**.

Строковые инструкции автоматически модифицируют содержимое регистров **SI** и/или **DI** для обеспечения возможности обработки следующего элемента строки. Значение флага направления **DF** определяет, будут ли эти индексные регистры автоматически увеличиваться (**DF=0**) или автоматически уменьшаться (**DF=1**) при переходе к следующему элементу строки. При обработке строк байтов содержимое **SI** и/или **DI** изменяется на 1; в случае строк слов — на 2.

Использование регистров и флагов строковыми инструкциями:

- **SI —** смещение строки-источника;
- **DI —** смещение строки-приемника;
- **CX** — счетчик повторений;
- **AL/AX** — значение сканирования: приемник для **LODS**, источник для **STOS**;
- **DF 0** — автоматическое увеличение **SI;**
- **DI 1** — автоматическое уменьшение **SI**;
- **ZF** — признак прекращения сканирования/сравнения.

При использовании префикса повторений содержимое регистра **CX** уменьшается на 1 после каждого повторения строковой инструкции. Регистр **CX** должен быть загружен требуемым числом повторений до выполнений строковой операции. Если **CX** содержит 0, строковая операция не выполняется, и управление передается следующей инструкции.

REP/REPE/REPZ/REPNE/REPNZ

Префиксы повторения. Эти ключевые слова представляют собой 5 мнемоник 2-х форм однобайтного префикса, управляющего повторением непосредственно следующей за ним строковой инструкции. Различные мнемоники введены для удобства программирования. Наличие префикса на состояния флагов не влияет.

REP используется в сочетании с инструкциями **MOVS** и **STOS** и интерпретируется как «повторение пока не конец строки» (в **CX** не 0). **REPE** и **REPZ** работают также и физически являются тем же префиксом, что и **REP**. **REPE** и **REPZ** используются в сочетании с инструкциями **CMPS** и **SCAS** и требуют, чтобы флаг **ZF**, используемый этими инструкциями, был установлен в 1 до инициализации следующего повторения.

REPNE и **REPNZ** представляют собой 2 мнемоники одного префикса и функционируют также, как **REPE** и **REPF**, но флаг **ZF** должен быть установлен в 0, или повторение прекратится. Заметим, что устанавливать флаг **ZF** перед выполнением повторяемой строковой инструкции необязательно.

Повторяемая строковая инструкция может быть прервана (на обработку системных прерываний это не распространяется). Процессор распознает это прерывание до обработки очередного элемента строки. После возврата из прерывания повторяемая операция возобновляется с точки прерывания.

Заметим, однако, что выполнение не возобновится правильно, если в дополнение к префиксу повторения был специфицирован 2-й или 3-й префикс (например, переключения сегмента или **LOCK**). В момент прерывания процессор «запоминает» только один префикс, причем, тот, который при кодировании операции непосредственно предшествовал строковой инструкции.

После возврата из прерывания остальные префиксы действовать не будут. Если все же необходимо воздействие более, чем одного, префикса, прерывания на время работы строковой операции следует запретить инструкцией **CLI**. Это, однако не поможет при появлении немаскируемого прерывания.

Кроме того, при обработке длинных строк может недопустимо возрасти время, в течении которого система не сможет отвечать на прерывания.



Пересылка строк

MOVS приемник,источник

Пересылка строки байтов или слов. Эта инструкция пересылает байт или слово источника, адресуемого регистром **SI**, в строку-приемник, адресуемую регистром **DI**, и модифицирует содержимое регистров **SI** и **DI** таким образом, чтобы они указывали на следующие элементы строк. Величина элементов строк и соответственно тип пересылки (байт или слово) определяется Ассемблером путем анализа атрибутов операндов инструкции. При использовании префикса **REP** инструкция **MOVS** может пересылать блоки памяти.

MOVSB/MOVSW

Пересылка строки байтов или слов. Эти инструкции обеспечивают пересылку байта (**MOVSB**) или слова (**MOVSW**) из элемента строки-источника, адресуемого регистром **SI**, в элемент строки-приемника, адресуемого регистром **DI**. Содержимое регистров **SI** и **DI** изменяется (уменьшается или увеличивается в соответствии со значением флага **DF**) на 1 для **MOVSB** или на 2 для **MOVSW** с тем, чтобы они указывали на следующие элементы строк. Использование этих инструкций полезно в том случае, когда Ассемблер не может определить атрибуты строк, например, при пересылке участка программного кода. Эти инструкции могут повторяться при использовании соответствующих префиксов.

LODS источник

Загрузка строки байтов или слов. Инструкция **LODS** загружает элемент строки-источника (байт или слово в зависимости от типа операнда), адресуемый регистром **SI**, в регистр **AL** или **AX** соответственно и устанавливает **SI** указывающим на следующий элемент строки. Обычно эта инструкция не повторяется, так как каждое повторение замещало бы содержимое регистров **AL** или **AX**, и сохранялось бы только последнее значение.

Однако, инструкция **LODS** весьма полезна в программных циклах как часть более сложной строковой операции.

LODSB/LODSW

Загрузка строки байтов или слов. Работа этих инструкций аналогична **LODS** с той лишь разницей, что здесь длина элемента строки задана явно: 1 байт для **LODSB** и 2 байта для **LODSW**.

STOS приемник

Сохранение строки байтов или слов. Инструкция **STOS** помещает содержимое регистров **AL** или **AX** (в зависимости от типа операнда) в элемент строки-приемника, адресуемый регистром **DI**, и устанавливает регистр **DI** указывающим на следующий элемент строки. Как повторяемая инструкция **STOS** является традиционным средством для заполнения строки каким-либо значением.

STOSB/STOSW

Сохранение строки байтов или слов. Работа этих инструкций аналогична **STOS** с той лишь разницей, что здесь длина элемента строки задана явно: 1 байт для **STOSB** и 2 байта для **STOSW**.



Сравнение строк

CMPS приемник,источник

Сравнение строки байтов или слов. Инструкция **CMPS** вычитает байт или слово строки-приемника, адресуемые регистром **DI**, из байта или слова строки-источника, адресуемых регистром **SI**. Величина элементов строк определяется Ассемблером путем анализа атрибутов операндов инструкции. **CMPS** не изменяет содержимое самих строк, но устанавливает флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF** таким образом, что они отражают отношение элемента строки-приемника к элементу строки-источника. Если инструкция **CMPS** использована с префиксом **REPE** или **REPZ**, выполняется операция «сравнение до конца строки (пока в CX не 0) и пока строки равны (**ZF=1**)». Если **CMPS** использована с префиксом **REPNE** или **REPNZ**, выполняется операция «сравнение до конца строки (пока в **CX** не 0) и пока строки не равны (**ZF=0**)». Таким образом, инструкция **CMPS** может применяться для поиска совпадающих или несовпадающих элементов строк.

CMPSB/CMPSW

Сравнение строки байтов или слов. Работа этих инструкций аналогична **CMPS** с той лишь разницей, что здесь длина элемента строк задана явно: 1 байт для **CMPSB** и 2 байта для **CMPSW**.



Сканирование

SCAS приемник

Сканирование строки байтов или слов. Инструкция **SCAS** вычитает элемент строки-приемника (байт или слово в зависимости от типа операнда), адресуемый регистром **DI**, из содержимого регистра **AL** или **AX** соответственно и модифицирует флаги, но не меняет ни строку, ни содержимое аккумулятора. После **SCAS** регистр **DI** указывает на следующий элемент строки, а флаги **AF**, **CF**, **OF**, **PF**, **SF** и **ZF** отражают отношение содержимого аккумулятора к элементу строки. Если присутствует префикс **REPE** или **REPZ**, выполняется операция «сканирование до конца строки (пока в **CX** не 0) и пока элемент строки равен содержимому аккумулятора (**ZF=1**)». Если присутствует префикс **REPNE** или **REPNZ**, выполняется операция «сканирование до конца строки (пока в **CX** не 0) и пока элемент строки не равен содержимому аккумулятора (**ZF=0**)». Этот способ может использоваться для поиска значения в строке.

SCASB/SCASW

Сканирование строки байтов или слов. Работа этих инструкций аналогична **SCAS** с той лишь разницей, что здесь длина элемента строки задана явно: 1 байт для **SCASB** и 2 байта для **SCASW**.

Практикум



Двупросмотровый алгоритм

Макропроцессор, как и язык Ассемблера, просматривает и обрабатывает строки текста. Но в языке все строки связаны адресацией — одна строка может ссылаться на другую при помощи адреса или имени, которое должно быть «известно» Ассемблеру.

Более того, адрес присваиваемый каждой отдельной строке зависит от содержимого, количества и адресов предшествующих строк. Если рассматривать макроопределение, как единый объект, то можно сказать, что строки нашего макроопределения не так сильно взаимосвязаны.

Макроопределения не могут ссылаться на объекты вовне этого макроопределения.

Предположим, что в теле макроопределения есть строка INCR X, причем перед этой командой параметр X получил значение 10. Макропроцессор не производит синтаксический анализ, а производит простую текстовую подстановку вместо «X» подставляется «10».

Наш алгоритм будет выполнять 2 систематических просмотра входного текста. В первый проход будут детерминированы все макроопределения, во второй проход будут открыты все ссылки на макросы.

Так же, как и язык Ассемблера не может выполнить ссылку на символ до того момента, как он встретит этот символ, язык макрокоманд не может выполнить расширение до тех пор, пока не встретит соответствующее макроопределение.

Во время первого просмотра проверяется каждый код операции, макроопределения запоминаются в таблице макроопределений, а копия исходного текста без макроопределений запоминается во внешней памяти, для использования ее при втором проходе.

Помимо таблицы макроопределений во время первого прохода будет также таблица имен, во второй проход она будет использоваться

для выделения макроопераций и расширения их до текста соответствующего макроопределения.

Данные для первого просмотра

```
1. ВХТ – Входной текст
2. ВЫХ1 – Выходная копия текста для использования во второй проход.
3. МДТ – таблица макроопределений, в которой хранятся тела макроопределений
4. МНТ – таблица имен, необходимая для хранения имен макрокоманд, определенных в МНТ
5. МДТС – счетчик для таблицы МДТ
6. МНТС – счетчик для таблицы МНТ
7. АЛА – массив списка параметров для подстановки индексных маркеров вместо формальных параметров, перед запоминанием определения.
```

Данные для второго просмотра

```
1. ВЫХ1 – Выходная копия текста после первого прохода
2. ВЫХ2 – Выходная копия текста после второго прохода
3. МДТ – таблица макроопределений, в которой хранятся тела макроопределений
4. МНТ – таблица имен, необходимая для хранения имен макрокоманд, определенных в МНТ
5. МДТС – счетчик для таблицы МДТ
6. МНТС – счетчик для таблицы МНТ
7. АЛА – массив списка параметров для подстановки индексных маркеров вместо формальных параметров, перед запоминанием определения.
```

Алгоритм

Приведем формальную запись соответствующих алгоритмов обработки макроопределений двухпросмотровым способом.

Каждый из алгоритмов осуществляет построчный просмотр входного текста.

Первый просмотр — макроопределения

Алгоритм первого просмотра проверяет каждую строку входного текста. Если она представляет собой псевдооперацию MACRO, то все следующие за ней строки запоминаются в ближайших свободных ячейках МДТ.

Первая строка макроопределения — это имя самого макроса. Имя заносится в таблицу имен МНТ с индексом этой строки в МДТ. При

этом происходит также подстановка номеров формальных параметров, вместо их имен.

Если в течение просмотра встречается команда END, то это означает, что весь текст обработан, и управление можно передавать второму просмотру для обработки макрокоманд.

Второй просмотр — расширение макрокоманд

Алгоритм второго просмотра проверяет мнемонический код каждого предложения. Если это имя содержится в МНТ, то происходит обработка макропредложения по следующему правилу: из таблицы МНТ берется указатель на начало описания макроса в МДТ.

Макропроцессор готовит массив списка АЛА содержащий таблицу индексов формальных параметров и соответствующих операндов макрокоманды. Чтение производится из МДТ, после чего в прочитанную строку подставляются необходимые параметры, и полученная таким образом строка записывается в ВЫХТ2. Когда встречается директива END, текст полученного кода передается для компиляции Ассемблеру.

Первый просмотр

```
Начало алгоритма
  МДТС = 0
  МНТС = 0
  ФЛАГ ВЫХОДА = 0

  цикл пока (ФЛАГ ВЫХОДА == 0) {

          чтение следующей строки ВХТ

          если !(операция MACRO) {
                  вывод строки в ВЫХТ1
                  если (операция END) ФЛАГ ВЫХОДА = 1
          }
          иначе {
                  чтение идентификатора
                  запись имени и индекса в МНТ
                  МНТС ++
                  приготовить массив списка АЛА
                  запись имени в МДТ
                  МДТС ++

                  цикл {
                          чтение следующей строки ВХТ
```



```
                          подстановка индекса операторов
                          добавление в МДТ
                          МДТС ++
                  } пока !(операция MEND)
          }
  }

  переход ко второму проходу
конец алгоритма
```

Второй просмотр

```
Начало алгоритма

  ФЛАГ ВЫХОДА = 0

  цикл пока (ФЛАГ ВЫХОДА == 0) {
          чтение строки из ВЫХТ1
          НАЙДЕНО = поиск кода в МНТ

          если !(НАЙДЕНО) {
                  запись в ВЫХТ2 строки

                  если (операция END) {
                          ФЛАГ ВЫХОДА = 1
                  }
          }

          иначе {
                  УКАЗАТЕЛЬ = индекс из МНТ
                  Заполнение списка параметров АЛА
                  цикл {
                          УКАЗАТЕЛЬ ++
                          чтение след. строки из МДТ
                          подстановка параметров
                          вывод в ВЫХТ2
                  } пока !(операция MEND)
          }
  }

  переход к компиляции

конец алгоритма
```



Однопросмотровый алгоритм

Предположим, что мы допускаем реализацию макроопределения внутри макроопределений. Основная проблема здесь заключена в том, что внутреннее макро определено только после того, как выполнен вызов внешнего.

Для обеспечения использования внутреннего макро нам придется повторять как просмотр обработки макроопределений, так и просмотр обработки макрокоманд. Однако существует и еще одно решение, которое позволяет произвести распознавание и расширение в один просмотр.

Рассмотрим аналогию с Ассемблером. Макроопределение должно обрабатываться до обработки макрокоманд, поскольку макро должны быть определены для процессора раньше, чем макрокоманды обращения к ним.

Однако, если мы наложим ограничение, что каждое макроопределение должно быть определено до того, как произойдет обращение к нему, мы устраним основное препятствие для однопросмотровой обработки.

Заметим, что то же самое может быть верно и для символических имен в Ассемблере, но такое требование было бы неоправданным ограничением для программиста.

В случае же макрорасширения может быть вполне естественно потребовать, чтобы объявления макро предшествовали вызовам. Это не накладывает очень существенных ограничений на использование аппарата макрокоманд.

Этот механизм даже не запрещает обращение макро к самому себе, поскольку обращение ведется в тот момент, когда имя макроса уже определено.

Расширение же макроса идет не в процессе разбора макроса, а в процессе последующего вызова.

Предложенный ниже алгоритм объединяет два вышеприведенных алгоритма для двупросмотрового макроассемблера в один.

Алгоритм

Однопросмотровый макроассемблер

```
Начало алгоритма

 МТДС = 0
 МНТС = 0
 ФЛАГ ВЫХОДА = 0

 цикл пока !(ФЛАГ ВЫХОДА) {

        чтение следующей строки ВХТ
        НАЙДЕНО = поиск кода в МНТ
        если (НАЙДЕНО) {

                МДИ = 1
                УКАЗАТЕЛЬ = индекс из МНТ
                Заполнение списка параметров АЛА

                цикл {
                        УКАЗАТЕЛЬ ++
                        чтение след. строки из МДТ
                        подстановка параметров
                        вставка во ВХТ
                } пока !(операция MEND)


        иначе если !(операция MACRO) {
                вывод строки в ВЫХТ1
                если (операция END) ФЛАГ ВЫХОДА = 1
        }
        иначе {
                чтение идентификатора
                запись имени и индекса в МНТ
                МНТС ++
                приготовить массив списка АЛА
                запись имени в МДТ
                МДТС ++

                цикл {
                        чтение следующей строки ВХТ
                        подстановка индекса операторов
                        добавление в МДТ
                        МДТС ++
```

```
                    } пока !(операция MEND)
                }
            }

        конец алгоритма
```

Описание алгоритма

Данный алгоритм является упрощением алгоритма приведенного раньше. Различие состоит в том, что современные средства интеллектуализации программирования дают нам возможность осуществлять вставки и удаления из крупных массивов с минимальными затратами процессорного времени, что было невозможно при использовании перфокарт.

Кроме того, скорость работы современных процессоров настолько велика, что позволяет производить прямые вставки и удаления в массивах данных средней величины (скажем, до 64 килобайт) в режиме реального времени.

Таким образом, расширение исходного макроса может быть напрямую вставлено в массив исходного текста и обработано в расширенном виде. Такая технология позволяет значительно упростить алгоритм обработки макроязыка.



Реализация внутри Ассемблера

Макропроцессор, описанный нами предназначался для обработки текста в режиме препроцессора, то есть он выполнял полный просмотр входного текста, до того, как передать управление Ассемблеру.

Но макропроцессор также может быть реализован внутри первого прохода Ассемблера. Такая реализация позволяет исключить промежуточные файлы, и позволяет достичь на порядок большей интеграции макропроцессора и Ассемблера путем объединения сходных функций.

Например, возможно объединение таблиц имен макросов и имен кода операции; специальный признак может указывать на то макро это или встроенная операция.

Основные преимущества включения макропроцессора в первый просмотр состоят в следующем:

- Многие функции не надо реализовывать дважды (например, функции ввода-вывода, проверки на тип, и т.п.)
- В процессе обработки отпадает необходимость создавать промежуточные файлы или массивы данных.
- У программиста появляются дополнительные возможности по совмещению средств Ассемблера (например, команды EUQ) совместно с макрокомандами.

Основные недостатки:

- Программа должна требовать больше оперативной памяти, что критично на некоторых типах ЭВМ, не имеющих много оперативной памяти.
- Реализация подобного типа задачи может оказаться на порядок сложнее, чем отдельная реализация Ассемблера и макропроцессора.

Отдельно от рассмотрения реализации аппарата макросредств в Ассемблер лежит рассмотрение дополнительного просмотра, используемого многими программами для выявления определенных характеристик исходной программы, таких как типы данных.

Располагая таким макропроцессором, можно использовать команды условной компиляции, позволяющие поставить расширение макрокоманд в зависимость от определенных характеристик программы.

Лабораторные работы



Общие указания к выполнению

Цель лабораторного практикума

Лабораторный практикум выполняется при изучении курса «Системное программирование и операционные системы» и имеет целью выработку у студентов навыков в трех направлениях:

- применение языка программирования С как инструмента для системного программирования;
- программное управление аппаратными средствами ПЭВМ на низком уровне;
- использование внутренних структур данных операционной системы MS DOS для получение информации и оперативной настройки системы.



Язык С как инструмент системного программирования

Главным качеством языка С, которое делает его именно языком системного программиста, является то, что «С — это язык относительно «низкого уровня»... Это означает, что С имеет дело с объектами того же вида, что и большинство ЭВМ, а именно, с символами, числами и адресами. Они могут объединяться и пересылаться посредством обычных арифметических и логических операций, осуществляемых реальными ЭВМ.» [2]. Система программирования С при представлении данных не

вносит никаких дополнительных структур памяти, которые не были бы «видны» программисту. Так, например, внутреннее представление массивов в языке С полностью совпадает с внешним: массив — это только последовательность слотов в памяти. Отсутствие специального дескриптора массива, с одной стороны, делает невозможным контроль выхода индексов за допустимое границы, но с другой, уменьшает объем памяти программы и увеличивает ее быстродействие за счет отсутствия в памяти дескриптора и команд проверки индекса при каждом обращении к элементу массива. Это общий принцип С-программ: программист имеет максимальные возможности для разработки эффективных программ, но вся ответственность за их правильность ложится на него самого. Поэтому отладка программ на языке С — непростой процесс, С-программы чаще «зависают» и выдают результаты, которые не всегда воспроизводятся и которые труднее объяснить, чем программы на других языках.

Чрезвычайно важным свойством языка С, которого нет в других языках, является адресная арифметика. Над данными типа «указатель» возможны арифметические операции, причем в них могут учитываться размеры тех объектов, которые адресуются указателем. Другое свойство указателей — их явная связь с конструкциями интеграции данных (массивы, структуры, объединения) и возможность подмены операций индексации и квалификации операциями адресной арифметики. За счет указателей программист имеет возможность удобным для себя способом структурировать адресное пространство программы и гибко изменять это структурирование.

Свойством, которое вытекает из общих принципов построения языка С, является слабая защита типов. В языках с сильной защитой типов (Pascal) для каждого типа данных определен набор доступных операций и компилятор запрещает применение к типу непредусмотренных для него операций и смешивание в выражениях данных разных типов. В С определен богатый набор правил преобразования типов по умолчанию, поэтому почти любая операция может быть применена к почти любому типу данных и выражения могут включать данные самых разных типов.

Еще некоторые средства языка не ориентированы непосредственно на низкоуровневое системное программирование, но могут быть очень полезны при разработке системных программ:

- обязательной составной частью языка является препроцессор. С не поддерживает сложных структур данных, но позволяет программисту определять свои типы. Включение в программу описания таких типов средствами препроцессора позволяет обеспечить

однозначную интерпретацию типов во всех модулях сложного программного изделия;

- процедурно-ориентированный язык С вместе с тем представляет все кодовые составляющие программы в виде функций. Это дает возможность применять язык С и як инструмент функционально-ориентированного программирования, как, например, язык LISP.

Важное качество языка С — высокая эффективность объектных кодов С-программы как по быстродействию, так и по объему памяти. Хотя это качество обеспечивается не столько свойствами самого языка, сколько свойствами системы программирования, традиция построения систем программирования С именно такова, что они обеспечивают большую, эффективность, чем, например, Pascal. Это связано еще и с «родословной» языков. Pascal возник как алгоритмический язык, предназначенный в своей первой версии не для написания программ, а для описания алгоритмов. Отсюда Pascal-трансляторы строились и строятся как синтаксически-ориентированные трансляторы характерно компилирующего типа: транслятор выполняет синтаксический анализ программы в соответствии с формально представленными правилами, а объектные коды формируются в основном в виде обращений к библиотечным процедурам, которые реализуют элементарные функции. Язык С ведет свое происхождение от языка BCPL, который был языком Макроассемблера. Отсюда и объектный код С-программы строится як последовательность машинных команд, которая оптимизируется для каждого конкретного выполнения данной функции.



Порядок выполнения работ

Для выполнения всех лабораторных работ предлагается единый порядок, предусматривающий следующие шаги.

- Ознакомиться с постановкой задачи и исходными данными. Определить вариант индивидуального задания.
- Сконструировать структуру программы.
- Составить текст программы.

- Набрать текст программы.
- Выполнить компиляцию программы.
- Провести анализ и исправление обнаруженных синтаксических ошибок в тексте программы.
- Получить решение (изображение) и, в случае обнаружения логических ошибок, определить и устранить их.



Содержание отчета

Отчет оформляется по каждой лабораторной работе и состоит из следующих разделов.

- Лекция лабораторной работы.
- Цель работы.
- Индивидуальное задание.
- Описание структур данных и алгоритмов
- Результаты работы программы.
- Интерпретация результатов.



Лабораторная работа №1. Работа с символьными строками

Цель работы

Получение практических навыков в работе с массивами и указателями языка С, обеспечение функциональной модульности

Темы для предварительного изучения

- Указатели в языке С.
- Представление строк.
- Функции и передача параметров.

Постановка задачи

По индивидуальному заданию создать функцию для обработки символьных строк. За образец брать библиотечные функции обработки строк языка С, но не применять их в своей функции. Предусмотреть обработку ошибок в задании параметров и особые случаи. Разработать два варианта заданной функции — используя традиционную обработку массивов и используя адресную арифметику.

Индивидуальные задания

1. Функция Copies(s,s1,n)

Назначение: копирование строки s в строку s1 n раз

2. Функция Words(s)

Назначение: подсчет слов в строке s

3. Функция Concat(s1,s2)

Назначение: конкатенация строк s1 и s2 (аналогичная библиотечная функция С — strcat)

4. Функция Parse(s,t)

Назначение: разделение строки s на две части: до первого вхождения символа t и после него

5. Функция Center(s1,s2,l)

Назначение: центрирование — размещение строки s1 в середине строки s2 длиной l

6. Функция Delete(s,n,l)

Назначение: удаление из строки s подстроки, начиная с позиции n, длиной l (аналогичная библиотечная Функция есть в Pascal).

7. Функция Left(s,l)

Назначение: выравнивание строки s по левому краю до длины l.

8. Функция Right(s,l)

Назначение: выравнивание строки s по правому краю до длины l.

9. Функция Insert(s,s1,n)

Назначение: вставка в строку s подстроки s1, начиная с позиции n (аналогичная библиотечная функция есть в Pascal).

10. Функция Reverse(s)

Назначение: изменение порядка символов в строке s на противоположный.

11. Функция Pos(s,s1)

Назначение: поиск первого вхождения подстроки s1 в строку s (аналогичная функция С — strstr).

12. Функция LastPos(s,s1)

Назначение: поиск последнего вхождения подстроки s1 в строку s.

13. Функция WordIndex(s,n)

Назначение: определение позиции начала в строке s слова с номером n.

14. Функция WordLength(s,n)

Назначение: определение длины слова с номером n в строке s.

15. Функция SubWord(s,n,l)

Назначение: выделение из строки s l слов, начиная со слова с номером n.

16. Функция WordCmp(s1,s2)

Назначение: сравнение строк (с игнорированием множественных пробелов).

17. Функция StrSpn(s,s1)

Назначение: определение длины той части строки s, которая содержит только символы из строки s1.

18. Функция StrCSpn(s,s1)

Назначение: определение длины той части строки s, которая не содержит символы из строки s1.

19. Функция Overlay(s,s1,n)

Назначение: перекрытие части строки s, начиная с позиции n, строкой s1.

20. Функция Replace(s,s1,s2)

Назначение: замена в строке s комбинации символов s1 на s2.



21. Функция Compress(s,t)

Назначение: замена в строке s множественных вхождений символа t на одно.

22. Функция Trim(s)

Назначение: удаление начальных и конечных пробелов в строке s.

23. Функция StrSet(s,n,l,t)

Назначение: установка l символов строки s, начиная с позиции n, в значение t.

23. Функция Space(s,l)

Назначение: доведение строки s до длины l путем вставки пробелов между словами.

24. Функция Findwords(s,s1)

Назначение: поиск вхождения в строку s заданной фразы (последовательности слов) s1.

25. Функция StrType(s)

Назначение: определение типа строки s (возможные типы — строка букв, десятичное число, 16-ричное число, двоичное число и т.д.).

26. Функция Compul(s1,s2)

Назначение: сравнение строк s1 и та s2 с игнорированием различий в регистрах.

27.Функция Translate(s,s1,s2)

Назначение: перевод в строке s символов, которые входят в алфавит s1, в символы, которые входят в алфавит s2.

28. Функция Word(s)

Назначение: выделение первого слова из строки s.

Примечание: под «словом» везде понимается последовательность символов, которая не содержит пробелов.



Пример решения задачи

Индивидуальное задание

Функция substr(s,n,l)

Назначение: выделение из строки s подстроки, начиная с позиции n, длиной l.

Описание метода решения

Символьная строка в языке C представляется в памяти как массив символов, в конце которого находится байт с кодом 0 — признак конца строки. Строку, как и любой другой массив можно обрабатывать либо традиционным методом — как массив, с использованием операции индексации, либо через указатели, с использованием операций адресной арифметики. При работе со строкой как с массивом нужно иметь в виду, что длина строки заранее неизвестна, так что циклы должны быть организованы не со счетчиком, а до появления признака конца строки.

Функция должна реализовывать поставленную задачу — и ничего более. Это означает, что функцию можно будет, например, перенести без изменений в любую другую программу, если спецификации функции удовлетворяют условиям задачи. Это также означает, что при ошибочном задании параметров или при каких-то особых случаях в их значениях функция не должна аварийно завершать программу или выводить какие-то сообщения на экран, но должна возвращать какое-то прогнозируемое значение, по которому та функция, которая вызвала нашу, может сделать вывод об ошибке или об особом случае.

Определим состав параметров функции:

```
int substr (src, dest, num, len);
```

где

- **src** — строка, с которой выбираются символы;
- **dest** — строка, в которую записываются символы;
- **num** — номер первого символа в строке src, с которого начинается подстрока (нумерация символов ведется с 0);
- **len** — длина выходной строки.

Возможные возвращаемые значения функции установим: 1 (задание параметров правильное) и 0 (задание не правильное). Эти значения при обращениях к функции можно будет интерпретировать как «истина» или «ложь».

Обозначим через **Lsrc** длину строки **src**. Тогда возможны такие варианты при задании параметров:

- **num+len <= Lsrc** — полностью правильное задание;
- **num+len > Lsrc; num < Lsrc** — правильное задание, но длина выходной строки будет меньше, чем **len**;
- **num >= Lsrc** — неправильное задание, выходная строка будет пустой;
- **num < 0** или **len <= 0** — неправильное задание, выходная строка будет пустой.

Заметим, что интерпретация конфигурации параметров как правильная/неправильная и выбор реакции на неправильное задание — дело исполнителя. Но исполнитель должен строго выполнять принятые правила. Возможен также случай, когда выходная строка выйдет большей длины, чем для нее отведено места в памяти. Однако, поскольку нашей функции неизвестен размер памяти, отведенный для строки, функция не может распознать и обработать этот случай — так же ведут себя и библиотечные функции языка С.

Описание логической структуры

Программа состоит из одного программного модуля — файл **LAB1.C**. В состав модуля входят три функции — **main**, **substr_mas** и **subs_ptr**. Общих переменных в программе нет. Макроконстантой **N** определена максимальная длина строки — 80.

Функция **main** является главной функцией программы, она предназначена для ввода исходных данных, вызова других функций и вывода результатов. В функции определены переменные:

- **ss** и **dd** — входная и выходная строки соответственно;
- **n** — номер символа, с которого должна начинаться выходная строка;
- l — длина выходной строки.

Функция запрашивает и вводит значение входной строки, номера символа и длины. Далее функция вызывает функцию **substr_mas**, передавая ей как параметры введенные значения. Если функция **substr_mas** возвращает 1, выводится на экран входная и выходная строки, если 0 — выводится сообщение об ошибке и входная строка. Потом входная строка делается пустой и то же самое выполняется для функции **substr_ptr**.

Функция **substr_mas** выполняет поставленное задание методом массивов. Ее параметры: — **src** и **dest** — входная и выходная строки соответственно, представленные в виде массивов неопределенного размера; **num** и **len**. Внутренние переменные **i** и **j** используются как индексы в массивах.

Функция проверяет значения параметров в соответствии со случаем 4, если условия этого случая обнаружены, в первый элемент массива **dest** записывается признак конца строки и функция возвращает 0.

Если случай 4 не выявлен, функция просматривает num первых символов входной строки. Если при этом будет найден признак конца строки, это — случай 3, при этом в первый элемент массива dest записывается признак конца строки и функция возвращает 0.

Если признак конца в первых num символах не найден, выполняется цикл, в котором индекс входного массива начинает меняться от 1, а индекс выходного — от 0. В каждой итерации этого цикла один элемент входного массива пересылается в выходной. Если пересланный элемент является признаком конца строки (случай 2), то функция немедленно заканчивается, возвращая 1. Если в цикле не встретится конец строки, цикл завершится после len итераций. В этом случае в конец выходной строки записывается признак конца и Функция возвращает 1.

Функция **substr_ptr** выполняет поставленное задание методом указателей. Ее параметры: — **src** и **dest** — входная и выходная строки соответственно, представленные в виде указателей на начала строк; **num** и **len**.

Функция проверяет значения параметров в соответствии со случаем 4, если условия этого случая выявлены, по адресу, который задает **dest**, записывается признак конца строки и функция возвращает 0, эти действия выполняются одним оператором.

Если случай 4 не обнаружен, функция пропускает num первых символов входной строки. Это сделано циклом while, условием выхода из которого является уменьшение счетчика num до 0 или появление признака конца входной строки. Важно четко представлять порядок операций, которые выполняются в этом цикле:

- выбирается счетчик **num**;
- счетчик **num** уменьшается на 1;

- если выбранное значение счетчика было 0 — цикл завершается;
- если выбранное значение было не 0 — выбирается символ, на который указывает указатель **src**;
- указатель **src** увеличивается на 1;
- если выбранное значение символа было 0, то есть, признак конца строки, цикл завершается, иначе — повторяется.

После выхода из цикла проверяется значение счетчика **num**: если оно не 0, это означает, что выход из цикла произошел по признаку конца строки (случай 3), по адресу, который задает **dest**, записывается признак конца строки и функция возвращает 0.

Если признак конца не найден, выполняется цикл, подобный первому циклу **while**, но по счетчику **len**. В каждой итерации этого цикла символ, на который показывает **src** переписывается по адресу, задаваемому **dest**, после чего оба указателя увеличиваются на 1. Цикл закончится, когда будет переписано **len** символов или встретится признак конца строки. В любом варианте завершения цикла по текущему адресу, который содержится в указателе **dest**, записывается признак конца строки и функция завершается, возвращая 1.

Данные для тестирования

Тестирование должно обеспечить проверку работоспособности функций для всех вариантов входных данных. Входные данные, на которых должно проводиться тестирование, сведены в таблицу:

вариант	src	num	len	dest
1	012345	2	2	23
	012345	0	1	0
	012345	0	6	012345
2	012345	5	3	5
	012345	2	6	2345
	012345	0	7	012345
3	012345	8	2	пусто
4	012345	-1	2	пусто
	012345	5	0	пусто
	012345	5	-1	пусто

Текст программы

```
/*************************************************/
/******** Файл LAB1.C *************************/
#include <stdio.h>
```



```
#define N 80
/************************************************/
/* Функция выделения подстроки (массивы)   */
/***********************************************/
int substr_mas(char src[N],char dest[N],int num,int len){
  int i, j;
  /* проверка случая 4 */
  if ( (num<0)||(len<=0) ) {
   dest[0]=0; return 0;
   }
  /* выход на num-ый символ */
  for (i=0; i<=num; i++)
   /* проверка случая 3 */
   if ( src[i]=='\0') {
    dest[0]=0; return 0;
    }
  /* перезапись символов */
   for (i--, j=0; j<len; j++, i++) {
   dest[j]=src[i];
   /* проверка случая 2 */
   if ( dest[j]=='\0') return 1;
   }
 /* запись признака конца в выходную строку */
  dest[j]='\0';
  return 1;
 }
/***********************************************/
/* Функция выделение подстроки */
/*  (адресная арифметика) */
/**********************************************/
int substr_ptr(char *src, char *dest, int num, int len) {
  /* проверка случая 4 */
  if ( (num<0)||(len<=0) ) return dest[0]=0;
  /* выход на num-ый символ или на конец строки */
  while ( num-- && *src++ );
  /* проверка случая 3 */
  if ( !num ) return dest[0]=0;
  /* перезапись символов */
  while ( len-- && *src ) *dest++=*src++;
  /* запись признака конца в выходную строку */
  *dest=0;
  return 1;
 }
```

```
/***********************************************/
main()
 {
 char ss[N], dd[N];
 int n, l;
 clrscr();
 printf("Вводите строку:\n");
 gets(ss);
 printf("начало=");
 scanf("%d",&n);
 printf("длина=");
 scanf("%d",&l);
 printf("Массивы:\n");
 if (substr_mas(ss,dd,n,l)) printf(">>%s<<\n>>%s<<\n",ss,dd);
 else printf("Ошибка! >>%s<<\n",dd);
 dd[0]='\0';
 printf("Адресная арифметика:\n");
 if (substr_ptr(ss,dd,n,l)) printf(">>%s<<\n>>%s<<\n",ss,dd);
 else printf("Ошибка! >>%s<<\n",dd);
 getch();
 }
```



Лабораторная работа №2. Представление в памяти массивов и матриц

Цель работы

Получение практических навыков в использовании указателей и динамических объектов в языке С, создание модульных программ и обеспечение инкапсуляции.

Постановка задачи

Для разряженной матрицы целых чисел в соответствии с индивидуальным заданием создать модуль доступа к ней, у котором обеспечить экономию памяти при размещении данных.

Индивидуальные задания

1

Все нулевые элементы размещены в левой части матрицы

2

Все нулевые элементы размещены в правой части матрицы

3

Все нулевые элементы размещены выше главной диагонали

4

Все нулевые элементы размещены в верхней части матрицы

5

Все нулевые элементы размещены в нижней части матрицы

6

Все элементы нечетных строк — нулевые

7

Все элементы четных строк — нулевые

8

Все элементы нечетных столбцов — нулевые

9

Все элементы четных столбцов — нулевые

10

Все нулевые элементы размещены в шахматном порядке, начиная с 1-го элемента 1-й строки

11

Все нулевые элементы размещены в шахматном порядке, начиная со 2-го элемента 1-й строки

12

Все нулевые элементы размещены на местах с четными индексами строк и столбцов

13

Все нулевые элементы размещены на местах с нечетными индексами строк и столбцов

14

Все нулевые элементы размещены выше главной диагонали на нечетных строках и ниже главной диагонали — на четных

15

Все нулевые элементы размещены ниже главной диагонали на нечетных строках и выше главной диагонали — на четных

16

Все нулевые элементы размещены на главной диагонали, в первых 3 строках выше диагонали и в последних 3 строках ниже диагонали

17

Все нулевые элементы размещены на главной диагонали и в верхней половине участка выше диагонали

18

Все нулевые элементы размещены на главной диагонали и в нижней половине участка ниже диагонали

19

Все нулевые элементы размещены в верхней и нижней четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)

20

Все нулевые элементы размещены в левой и правой четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)

21

Все нулевые элементы размещены в левой и верхней четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)

22

Все нулевые элементы размещены на строках, индексы которых кратны 3

23

Все нулевые элементы размещены на столбцах, индексы которых кратны 3

24

Все нулевые элементы размещены на строках, индексы которых кратны 4

25

Все нулевые элементы размещены на столбцах, индексы которых кратны 4

26

Все нулевые элементы размещены попарно в шахматном порядке (сначала 2 нулевых)

27

Матрица поделена диагоналями на 4 треугольники, элементы верхнего и нижнего треугольников нулевые

28

Матрица поделена диагоналями на 4 треугольники, элементы левого и правого треугольников нулевые

29

Матрица поделена диагоналями на 4 треугольника, элементы правого и нижнего треугольников нулевые

30

Все нулевые элементы размещены квадратами 2x2 в шахматном порядке

Исполнителю самому надлежит выбрать, будут ли начинаться индексы в матрице с 0 или с 1.

Пример решения задачи

Индивидуальное задание:

- матрица содержит нули ниже главной диагонали;
- индексация начинается с 0.

Описание методов решения

Представление в памяти

Экономное использование памяти предусматривает, что для тех элементов матрицы, в которых наверняка содержатся нули, память выделяться не будет. Поскольку при этом нарушается двумерная структура матрицы, она может быть представлена в памяти как одномерный массив, но при обращении к элементам матрицы пользователь имеет возможность обращаться к элементу по двум индексам.

Модульная структура программного изделия

Программное изделие должно быть отдельным модулем, файл LAB2.C, в котором должны размещаться как данные (матрица и вспомогательная информация), так и функции, которые обеспечивают доступ. Внешний доступ к программам и данным модуля возможен только через вызов функций чтения и записи элементов матрицы. Доступные извне элементы программного модуля должны быть описаны в отдельном файле LAB2.H, который может включаться в программу пользователя оператором препроцессора:

```
#include "lab2.h"
```

Пользователю должен поставляться результат компиляции — файл LAB2.OBJ и файл LAB2.H.

Преобразование 2-компонентного адреса элемента матрицы, которую задает пользователь, в 1-компонентную должно выполняться отдельной функцией (так называемой, функцией линеаризации), вызов которой возможен только из функций модуля. Возможны три метода преобразования адреса:

- при создании матрицы для нее создается также и дескриптор D[N] — отдельный массив, каждый элемент которого соответствует одной строке матрицы; дескриптор заполняется значениями, подобранными так, чтобы: **n = D[x] + y**, где **x**, **y** — координаты пользователя (строка, столбец), **n** — линейная координата;
- линейная координата подсчитывается методом итерации как сумма полезных длин всех строк, предшествующих строке **x**, и к ней прибавляется смещение **y**-го полезного элемента относительно начала строки;
- для преобразования подбирается единое арифметическое выражение, которой реализует функцию: **n = f(x,y)**.

Первый вариант обеспечивает быстрейший доступ к элементу матрицы, ибо требует наименьших расчетов при каждом доступе, но плата за это — дополнительные затраты памяти на дескриптор. Второй вариант — наихудший по всем показателям, ибо каждый доступ требует выполнения оператора цикла, а это и медленно, и занимает память. Третий вариант может быть компромиссом, он не требует дополнительной памяти и работает быстрее, чем второй. Но выражение для линеаризации тут будет сложнее, чем первом варианте, следовательно, и вычисляться будет медленнее.

В программном примере, который мы приводим ниже, полностью реализован именно третий вариант, но далее мы показываем и существенные фрагменты программного кода для реализации и двух других.

Описание логической структуры

Общие переменные

В файле LAB2.C описаны такие статические переменные:

- **int NN** — размерность матрицы;
- **int SIZE —** количество ненулевых элементов в матрице;
- **int *m_addr —** адрес сжатой матрицы в памяти, начальное значение этой переменной — **NULL** — признак того, что память не выделена;
- **int L2_RESULT** — общий флаг ошибки, если после выполнения любой функции он равен -1, то произошла ошибка.

Переменные **SIZE** и **m_addr** описаны вне функций с квалификатором **static**, это означает, что вони доступны для всех функций в этом модуле, но недоступны для внешних модулей. Переменная **L2_RESULT** также описана вне всех функций, не без явного квалификатора. Эта переменная доступна не только для этого модуля, но и для всех внешних модулей, если она в них буде описана с квалификатором extern. Такое описание имеется в файле **LAB2.H**.

Функция creat_matr

Функция **creat_matr** предназначена для выделения в динамической памяти места для размещения сжатой матрицы. Прототип функции:

```
int creat_matr ( int N );
```

где **N** — размерность матрицы.

Функция сохраняет значение параметра в собственной статической переменной и подсчитывает необходимый размер памяти для размещения ненулевых элементов матрицы. Для выделения памяти используется библиотечная функция С **malloc**. Функция возвращает -1, если при выделении произошла ошибка, или 0, если выделение прошло нормально. При этом переменной **L2_RESULT** также присваивается значение 0 или -1.

Функция close_matr

Функция **close_matr** предназначена для освобождения памяти при завершении работы с матрицей,

Прототип функции:

```
int close_matr ( void );
```

Функция возвращает 0 при успешном освобождении, -1 — при попытке освободить невыделенную память.

Если адрес матрицы в памяти имеет значения **NULL**, это признак того, что память не выделялась, тогда функция возвращает -1, иначе — освобождает память при помощи библиотечной функции free и записывает адрес матрицы — **NULL**. Соответственно функция также устанавливает глобальный признак ошибки — **L2_RESULT**.

Функция read_matr

Функция **read_matr** предназначена для чтения элемента матрицы. Прототип функции:

```
int read_matr(int x, int y);
```

где **x** и **y** — координаты (строка и столбец). Функция возвращает значение соответствующего элемента матрицы. Если после выполнения функции значение переменной **L2_RESULT** -1, то это указывает на ошибку при обращении.

Проверка корректности задания координат выполняется обращением к функции **ch_coord**, если эта последняя возвращает ненулевое значение, выполнение **read_matr** на этом и заканчивается. Если же координаты заданы верно, то проверяется попадание заданного элемента в нулевой или ненулевой участок. Элемент находится в нулевом участке, если для него номер строки больше, чем номер столбца. Если элемент в нулевом участке, функция просто возвращает 0, иначе — вызывает функцию линеаризации lin и использует значение, которое возвращает **lin**, как индекс в массиве **m_addr**, по которому и выбирает то значения, которое возвращается.

Функция write_matr

Функция **write_matr** предназначена для записи элемента в матрицу. Прототип функции:

```
int write_matr(int x, int y, int value);
```

где **x** и **y** — координаты (строка и столбец), **value** — то значение, которое нужно записать. Функция возвращает значение параметра **value**, или 0 — если была попытка записи в нулевой участок. Если после выполнения функции значение переменной **L2_RESULT** -1, то это указывает на ошибку при обращении.

Выполнение функции подобно функции **read_matr** с тем отличием, что, если координаты указывают на ненулевой участок, то функция записывает **value** в массив **m_addr**.

Функция ch_coord

Функция **ch_coord** предназначена для проверки корректности задания координат. Эта функция описана как static и поэтому может вызываться только из этого же модуля. Прототип функции:

```
static char ch_coord(int x, int y);
```

где **x** и **y** — координаты (строка и столбец). Функция возвращает 0, если координаты верные, -1 — если неверные. Соответственно, функция также устанавливает значение глобальной переменной **L2_RESULT**.

Выполнение функции собственно состоит из проверки трех условий:

- адрес матрицы не должен быть **NULL**, то есть, матрица должна уже находиться в памяти;
- ни одна из координат не может быть меньше 0;
- ни одна из координат не может быть больше **NN**.

Если хотя бы одно из этих условий не выполняется, функция устанавливает признак ошибки.

Функция lin

Функция **lin** предназначена для преобразования двумерных координат в индекс в одномерном массиве. Эта функция описана как **static** и поэтому может вызываться только из этого же модуля. Прототип функции:

```
static int lin(int x, int y);
```

где **x** и **y** — координаты (строка и столбец). Функция возвращает координату в массиве **m_addr**.

Выражение, значение которого вычисляет и возвращает функция, подобрано вот из каких соображений. Пусть мы имеет такую матрицу, как показано ниже, и нам нужно найти линейную координату элемента, обозначенного буквой **A** с координатами (**x,y**):

```
x x x x x x
0 x x x x x
0 0 x x A x
0 0 0 x x x
0 0 0 0 x x
0 0 0 0 0 x
```

Координату элемента можно определить как:

```
n = SIZE - sizeX +offY,
```

где **SIZE** — общее количество элементов в матрице,

```
SIZE = NN * (NN - 1) / 2 + NN;
```

sizeX — количество ненулевых элементов, которые содержатся в строке **x** и ниже,

```
sizeX = (NN - x) * (NN - x - 1) / 2 + (NN - x);
```

offY — смещение нужного элемента от начала строки **x**,

```
offY = y - x
```

Программа пользователя

Для проверки функционирования нашего модуля создается программный модуль, который имитирует программу пользователя. Этот модуль обращается к функции **creat_matr** для создания матрицы нужного размера, заполняет ненулевую ее часть последовательно увеличивающимися числами, используя для этого функцию **write_matr**, и выводит матрицу на экран, используя для выборки ее элементов функцию **read_matr**. Далее в диалоговом режиме программа вводит запрос на свои действия и читает/пишет элементы матрицы с заданными координатами, обращаясь к функциям **read_matr/ write_matr**. Если пользователь захотел закончить работу, программа вызывает функцию **close_matr**.

Тексты программных модулей

```
/******* Файл LAB2.H *************************/
/*   Описание функций и внешних переменных файла LAB2.C   */
extern int L2_RESULT;  /* Глобальна переменная - флаг ошибки */
/***** Выделение памяти под матрицу */
int creat_matr ( int N );
/***** Чтение элемента матрицы по заданным координатам */
int read_matr ( int x, int y );
/***** Запись элемент в матрицу по заданным координатам */
int write_matr ( int x, int y, int value );
/***** Уничтожение матрицы */
int close_matr ( void );
/******* Конец файла LAB2.H *******************/
/********** Файл LAB2.C *************************/
/* В этом файле определены функции и переменные для обработки
матрицы, заполненной нулями ниже главной диагонали     */
#include <alloc.h>
static int NN;     /* Размерность матрицы */
static int SIZE;      /* Размер памяти */
```



```
static int *m_addr=NULL; /* Адрес сжатой
матрицы */
static int lin(int, int);   /* Описание функции линеаризации */
static char ch_coord(int, int);  /* Описание функции проверки */
int L2_RESULT;       /* Внешняя переменная, флаг ошибки */
/*********************************************/
/*  Выделение памяти под сжатую матрицу    */
int creat_matr ( int N ) {
  /* N - размер матрицы */
  NN=N;
  SIZE=N*(N-1)/2+N;
  if ((m_addr=(int *)malloc(SIZE*sizeof(int))) == NULL )
   return L2_RESULT=-1;
  else
   return L2_RESULT=0;
/* Возвращает 0, если выделение прошло успешно, иначе -1 */
}
/**********************************************/
/* Уничтожение матрицы (освобождение памяти)  */
int close_matr(void) {
  if ( m_addr!=NULL ) {
   free(m_addr);
   m_addr=NULL;
   return L2_RESULT=0;
   }
  else return L2_RESULT=-1;
/* Возвращает 0, если освобождение пршло успешно, иначе - -1 */
}
/*********************************************/
/*  Чтение элемента матрицы по заданным координатам   */
int read_matr(int x, int y) {
  /* x, y -координати (строка, столбец) */
  if ( ch_coord(x,y) ) return 0;

/* Если координаты попадают в нулевой участок - возвращается 0,
иначе - применяется функция линеаризации */
  return (x > y) ? 0 : m_addr[lin(x,y)];
/* Проверка успешности чтения - по переменной
L2_RESULT: 0 - без ошибок, -1 - была ошибка */
}

/*********************************************/
/*   Запись элемента матрицы по заданным координатам   */
```

```
int write_matr(int x, int y, int value) {
/* x, y -координати, value - записываемое значение */
  if ( chcoord(x,y) ) return;
  /* Если координаты попадают в нулевой участок - записи нет,
иначе - применяется функция линеаризации */
  if ( x > y ) return 0;
  else return m_addr[lin(x,y)]=value;
  /* Проверка успешности записи - по L2_RESULT */
}

/*************************************************/
/* Преобразование 2-мерных координат в линейную */
/*     (вариант 3)              */
static int lin(int x, int y) {
  int n;
  n=NN-x;
  return SIZE-n*(n-1)/2-n+y-x;
}

/*************************************************/
/*  Проверка корректности обращения       */
static char ch_coord(int x, int y) {
  if ( ( m_addr==NULL ) ||
    ( x>SIZE ) || ( y>SIZE ) || ( x<0 ) || ( y<0 ) )
/* Если матрица не размещена в памяти, или заданные координаты
выходят за пределы матрицы */
    return L2_RESULT=-1;
  return L2_RESULT=0;
}
/*******Конец файла LAB2.C ********************/
/******** Файл MAIN2.C *************************/
/* "Программа пользователя" */
#include "lab2.h"
main(){
 int R;  /* размерность */
 int i, j; /* номера строки и столбца */
 int m;  /* значения элемента */
 int op;  /* операция */
 clrscr();
 printf('Введите размерность матрицы >'); scanf("%d",R);
 /* создание матрицы */
 if ( creat_matr (R) ) {
   printf("Ошибка создания матрицы\n");
```



```
   exit(0);
   }
 /* заполнение матрицы */
 for ( m=j=0; j<R; j++)
   for ( i=o; i<R; i++)
     write_matr(i,j,++m);
  while(1) {
   /* вывод матрицы на экран */
   clrscr();
   for (j=0; j<R; j++) {
     for (i=0; i<R; i++)
      printf("%3d ",read_matr(i,j));
      printf("\n");
     }
   printf("0 - выход\n1 - чтение\n2 - запись\n>")
   scanf("%d",&op);
   switch(op) {
    case 0:
     if (close_matr()) printf("Ошибка при уничтожении\n");
     else printf("Матрица уничтожена\n");
     exit(0);
    case 1: case 2:
     printf("Введите номер строки >");
     scanf("%d",&j);
     printf("Введите номер столбца >");
     scanf("%d",&i);
     if (op==2) {
       printf("Введите значение элемента >");
       scanf("%d",&m);
       write_matr(j,i,m);
       if (L2_RESULT<0) pritnf("Ошибка записи\n");
       }
     else {
       m=read_matr(j,i);
       if (L2_RESULT<0) pritnf("Ошибка считывания\n");
       else printf("Считано: %d\n",m);
       }
     printf("Нажмите клавишу\n"); getch();
     break;
    }
   }
}
/*****Конец файла MAIN2.C **********************/
```

Варианты

Вариант 1 требует:

- добавления к общим статическим переменным еще переменной:

```
static int *D; /* адрес дескриптора */
```

- добавления такого блока в функцию **creat_matr**:

```
{
 int i, s;
 D=(int *)malloc(N*sizeof(int));
 for (D[0]=0,s=NN-1,i=1; i<NN; i++)
  D[i]=D[i-1]+s--;
}
```

- изменения функции **lin** на:

```
static int lin(int x, int y) {
  return D[x]+y;
}
```

Вариант 2 требует:

- изменения функции lin на:

```
static int lin(int x, int y) {
 int s;

  for (s=j=0; j<x; j++)
   s+=NN-j;
  return s+y-x;
}
```



Лабораторная работа №4. Проверка оборудования

Цель работы

Получение практических навыков в определении конфигурации и основных характеристик компьютера.

Постановка задачи

Для компьютера на своем рабочем месте определить:

- тип компьютера;
- конфигурацию оборудования;
- объем оперативной памяти;
- наличие и объем расширенной памяти;
- наличие дополнительных ПЗУ;
- версию операционной системы.

Пример решения задачи

Структура данных программы

Программа использует, так называемый, список оборудования — 2-байтное слово в области данных BIOS по адресу 0040:0010. Назначение разрядов списка оборудования такое:

- 0 установлен в 1, если есть НГМД
- 1 установлен в 1, если есть сопроцессор
- 2,3 число 16-Кбайтных блоков ОЗУ на системной плате
- 4,5 код видеоадаптера: 11 — MDA, 10 — CGA, 80 колонок, 01 — CGA, 40 колонок, 00 — другой
- 6,7 число НГМД-1 (если в разряде 0 единица)
- 8 9, если есть канал ПДП
- 9,10,11 число последовательных портов RS-232
- 12 1, если есть джойстик
- 13 1, если есть последовательный принтер
- 14,15 число параллельных принтеров

Структура программы

Программа состоит только из основной функции **main()**. Выделения фрагментов программы в отдельные процедуры не требуется, потому что нет таких операций, которые во время работы программы выполняются многократно.

Описание переменных

Переменные, применяемые в программе:

- **type_PC** — байт типа компьютера, записанный в ПЗУ BIOS по адресу FF00:0FFE;
- **a**, **b** — переменные для определения объема extended-памяти ПЭВМ, **a** — младший байт, **b** — старший байт;
- **konf_b** — 2-байтное слово из области данных BIOS, которое содержит список оборудования;
- **type** — массив символьных строк, представляющих типы компьютера;
- **typ1A** — массив байт, содержащий коды типов дисплеев;
- **types1A[] —** массив строк, содержащий названия типов дисплеев;
- **j** — вспомогательная переменная, которая используется для идентификации типа дисплея;
- **seg** — сегмент, в котором размещено дополнительное ПЗУ;
- **mark** — маркер ПЗУ;
- **bufVGA[64]** — буфер данных VGA, из которого (при наличии VGA) мы выбираем объем видеопамяти;
- **rr** и **sr** — переменные, которые используются для задания значения регистров общего назначения и сегментных регистров, соответственно, при вызове прерывания.

Описание алгоритма программы

Алгоритм основной программы может быть разбито на 5 частей.

Часть 1 предназначена для определения типа компьютера. Для этого прочитаем байт, записанный в ПЗУ BIOS по адресу FF00:0FFE. В зависимости от значения этого байта сделаем вывод о типе ПЭВМ. Так, например, компьютеру типа AT соответствует код 0xFC.

Часть 2 предназначена для определения конфигурации ПЭВМ. Для этого прочитаем из области данных BIOS список оборудования. Для определения количества дисководов (если бит 0 установлен в 1) необходимо выделить биты 6 и 7 (маска 00C0h) и сместить их вправо на 6 разрядов, а потом добавить 1.

Для определения количества 16-Кбайтных блоков ОЗУ на системной плате необходимо выделить биты 2 и 3 с помощью маски 000Ch, сместить вправо на 2 разряды и добавить 1.

Для определения количества последовательных портов RS-232 выделить с помощью маски 0Eh биты 9-11 и сместить вправо на 9 разрядов.

Для определения наличия математического сопроцессора — проверить установку бита 1 маской 0002h.

Для определения наличия джойстика — бита 12 с помощью маски 1000h.

Определить количество параллельных принтеров можно, выделив биты 14 и 15 маской C000h и сместив их вправо на 14 разрядов.

Поскольку список оборудования содержит недостаточно информации про дисплейный адаптер, то для уточнения типа адаптера выполним дополнительные действия.

Видеоадаптер обслуживается прерыванием BIOS 10h. Для новых типов адаптеров список его функций расширяется. Эти новые функции и используются для определения типу адаптера.

Функция 1Ah доступна только при наличии расширения BIOS, ориентированного на обслуживание VGA. В этом случае функция возвращает в регистре AL код 1Ah — свою «визитную карточку», а в BL — код активного видеоадаптера. В случае, если функция 1Ah поддерживается, обратимся еще к функции 1Bh — последняя заполняет 70-байтный блок информации про состояние, из которого мы выбираемо объем видеопамяти.

Если 1Ah не поддерживается, это означает, что VGA у нас нет, в этом случае можно обратиться к функции 12h — получение информации про EGA. При наличии расширения, ориентированного на EGA, эта функция изменяет содержимое BL (перед обращением он должен быть 10h) на 0 (цветной режим) или на 1 (монохромный режим) а в BH возвращает объем видеопамяти.

Если же ни 1Ah, ни 12 не поддерживаются, то список оборудования BIOS содержит достаточную информацию про видеоадаптер и, выделивши биты 4, 5 мы можем сделать окончательный вывод про тип адаптера, который у нас есть.

В третьей части программы определим объем оперативной памяти, наличие и объем extended-памяти. Объем оперативной памяти для AT может быть прочитан из регистров 15h (младший байт) и 16h (старший

байт) CMOS-памяти или из области памяти BIOS по адресу 0040:0013 (2-байтное слово). Кроме того, в ПЭВМ может быть еще и дополнительная (expanded) память свыше 1 Мбайту. Ее объем можно получит из регистров 17h (младший байт) и 18h (старший байт) CMOS-памяти. Для чтения регистра CMOS-памяти необходимо видать в порт 70h байт номера регистра, а потом из порта 71h прочитать байт содержимого этого регистра.

В следующей части программы определим наличие и объем дополнительных ПЗУ. В адресному пространстве от C000:0000 по F600:0000 размещаются расширения ПЗУ (эта память не обязательно присутствует в ПЭВМ). Для определения наличия дополнительного ПЗУ будем читать первое слово из каждых 2 Кбайт, начиная с адреса C000:0000 в поисках маркера расширения ПЗУ: 55AAh. Если такой маркер найден, то следующий байт содержит длину модуля ПЗУ.

В заключительной части программы определим версию DOS, установленную на ПЭВМ. Для этого воспользуемся функцией DOS 30h, которая возвращает в регистре AL старшее число номера версии, а в регистре AH — младшее число.

Текст программы

```
/*----Лабораторная работа N4-----------------*/
/*----"Проверка состава оборудования"--------*/

/* Подключение стандартных заголовков */
#include <dos.h>
#include <conio.h>
#include <stdio.h>

/*-----------------------------------------------*/
void main()
{
unsigned char type_PC, /* Тип компьютера  */
    a,b;   /* Переменные для определения */
         /* характеристик памяти ПЭВМ  */
unsigned int konf_b; /* Байт конфигурации из BIOS  */
 char *type[]={"AT","PCjr","XT","IBM PC","unknown"};
 unsigned char typ1A[]={0,1,2,4,5,6,7,8,10,11,12,0xff};
 char *types1A[]={"нема дисплею","MDA, моно","CGA, цв.",
  "EGA, цв.","EGA, моно","PGA, цв.",
  "VGA, моно, анал.","VGA, кол., анал.",
  "MCGA, кол., цифр.","MCGA, моно, анал."
  "MCGA, кол., анал.","неизвестный тип",
         "непредусмотренный код"};
```



```
unsigned int j;    /* Вспомогательная переменная */
unsigned int seg;   /* Сегмент ПЗУ    */
unsigned int mark=0xAA55;  /* Маркер ПЗУ  */
unsigned char bufVGA[64]; /* Буфер данных VGA */
union REGS rr;
struct SREGS sr;

textbackground(0);
clrscr();
textattr(0x0a);
cprintf("Лабораторная работа N5");
cprintf("\nПроверка состава оборудования");

/* Определения типа компьютера */
type_PC=peekb(0xF000,0xFFFE);
if( (type_PC-=0xFC)>4)
   type_PC=4;
textattr(0x0b);
cprintf("\nТип компьютера: ");
textattr(0x0f);
cprintf("%s\n\r",type[type_PC]);

/* Конфигурация*/
konf_b=peek(0x40,0x10); /* Чтение байта оборудования */
       /* из памяти BIOS      */
textattr(0x0b);
cprintf("Конфигурация:\n\r");


/* Количество дисководов */

textattr(0x0e);
cprintf("  Дисководов ГМД:      ");
textattr(0x0f);
if(konf_b&0x0001)
   cprintf("%d\n\r",((konf_b&0x00C0)>>6)+1);
else
 cprintf("нет\n\r");
textattr(0x0e);
cprintf("  Математич. сопроцессор:  ");
textattr(0x0f);
if(konf_b&0x0002)
 cprintf("есть\n\r");
```

```
else
 cprintf("нет\n\r");
textattr(0x0e);
cprintf("  Тип дисплейного адаптера: ");
textattr(0x0f);

/* Определение активного адаптера */
/* Предположим наличие VGA */
rr.h.ah=0x1a;
rr.h.al=0;
int86(0x10,&rr,&rr);
if(rr.h.al==0x1a) /* Поддерживается функция 1Ah */
{         /* прерывания 10h       */
for(j=0;j<12;j++)
 if(rr.h.bl==typ1A[j])
 break;
cprintf("%s",types1A[j]);

if(j>0 && j<12)
{
 rr.h.ah=0x1b;
 rr.x.bx=0;
 sr.es=FP_SEG(bufVGA);
 rr.x.di=FP_OFF(bufVGA);
 int86x(0x10,&rr,&rr,&sr);
 cprintf(", %d Кбайт\n\r",((int)bufVGA[49]+1)*64);
}
else
   cprintf("\n\r");
}
else
{
/* Предположим наличие EGA */
rr.h.ah=0x12;
rr.h.bl=0x10;
int86(0x10,&rr,&rr);
if(rr.h.bl!=0x10) /* Поддерживается функция 12h */
{         /* прерывания 10h */
 cprintf("EGA");
 if(rr.h.bh)
 cprintf(" моно");
 else
 cprintf(" кол.");
```



```
 cprintf(", %d Кбайт\n\r",((int)rr.h.bl+1)*64);
}
else
{
 /* CGA или MDA */
 switch(konf_b&0x0030)
 {
 case 0:  cprintf("EGA/VGA\n\r");break;
 case 0x10: cprintf("CGA,40\n\r");break;
 case 0x20: cprintf("CGA,80\n\r");break;
 case 0x30: cprintf("MDA");break;
 }
}
}

/* Блоки ОЗУ на системной плате */
textattr(0x0e);
cprintf("\n\r  Первичный блок памяти:  ");
textattr(0x0f);
switch (konf_b&0x000C)
{
case 0:cprintf("16 Кбайт\n\r");break;
case 4:cprintf("32 Кбайт\n\r");break;
case 8:cprintf("48 Кбайт\n\r");break;
case 12:cprintf("64 Кбайт или больше\n\r");break;
}

/* Количество последовательных портов RS-232 */
textattr(0x0e);

cprintf("  Портов RS232:        ");
textattr(0x0f);
cprintf("%d\n\r",(konf_b&0x0E00)>>9);

/* Наличие джойстика */
textattr(0x0e);
cprintf("  Джойстик:          ");
textattr(0x0f);
if(konf_b&0x1000 )
cprintf("есть\n\r");
else
cprintf("нет\n\r");
```

```
/* Количество параллельних принтеров */
textattr(0x0e);
cprintf("  Принтеров:        ");
textattr(0x0f);
cprintf("%d\n\n\r",(konf_b&0xC000)>>14);

/* Объем оперативной памяти */

textattr(0x0e);
cprintf("Объем оперативной памяти: ");
textattr(0x0f);
cprintf("%d Кбайт\n\r",peek(0x40,0x13));
textattr(0x0e);

/* Наличие и объем extended-памяти */
outportb(0x70,0x17);
a=inport(0x71);
outportb(0x70,0x18);
b=inport(0x71);
cprintf("Объем extended-памяти:  ");
textattr(0x0f);
cprintf("%d Кбайт\n\n\r",(b<<8)|a);

/* Наличие дополнительных ПЗУ */
for( seg=0xC000;seg<0xFFB0;seg+=0x40)
/* Просмотр памяти от C000:0 с шагом 2 К */
if(peek(seg,0)==mark)  /* Маркер найден */
{
 textattr(0x0a);
 cprintf("Адрес ПЗУ =");
 textattr(0x0f);
 cprintf(" %04x",seg);
 textattr(0x0a);
 cprintf(". Длина модуля = ");
 textattr(0x0f);
 cprintf("%d",512*peekb(seg,2));
 textattr(0x0a);

 cprintf(" байт\n\r",peekb(seg,2));
}

/* Определение версии операцийной системы */
rr.h.ah=0x30;
```



```
 intdos(&rr,&rr);
 textattr(0x0c);
 cprintf("\n\rВерсия MS-DOS ");
 textattr(0x0f);
 cprintf("%d.%d\n\r",rr.h.al,rr.h.ah);

 textattr(0x0a);
 gotoxy(30,24);
 cprintf("Нажмите любую клавишу");
 textattr(0x07);
 getch();
 clrscr();
}
```

Результаты работы программы

В процессе работы программы на экран была выведена такая информация:

```
Лабораторная работа N4
Проверка состава оборудования

Тип компьютера: AT
Конфигурация:
Дисководов ГМД:      2
Математич. сопроцессор:  есть
Тип дисплейного адаптера: VGA, кол., анал., 256 Кбайт
Первичный блок памяти:  16 Кбайт
Портов RS232:       2
Джойстик:          нет
Принтеров:         1
Объем оперативной памяти: 639 Кбайт
Объем extended-памяти:  384 Кбайт
Адрес ПЗУ = c000. Длина модуля = 24576 байт
Версия MS-DOS 6.20
```



Лабораторня работа №5. Управление клавиатурой

Цель работы

Изучение организации и принципов работы клавиатуры и закрепление практических навыков управления ею, а также практических навыков создания собственных программ обработки прерываний.

Постановка задачи

Разработать программу обработки прерывания от клавиатуры, которая должна:

- распознавать нажатие «горячей» комбинации клавиш и реагировать на него звуковым сигналом;
- при первом нажатии «горячей» комбинации переходить в режим блокировки ввода заданной клавиши, при втором — отменять этот режим;
- системная обработка всех других клавиш нарушаться не должна.

Индивидуальные задания

Пример решения задачи

Индивидуальное задание:

- комбинация клавиш **LeftCtrl+RightShift+F3**;
- блокирование ввода клавиши 3.

Разработка алгоритма

Структура программы

Программа состоит из основной программы и трех функций.

- **void *readvect(int in)** — функция читает вектор прерывания с номером **in** и возвращает его значение.

- **void writevect (int in, void *h)** — функция устанавливает новый вектор прерывания in на новый обработчик этого прерывания по адресу **h**.
- **void interrupt new9()** — процедура нового обработчика прерывания 9h.

Описание переменных

Глобальные переменные программы: **old9** — адрес старого обработчика прерывания 9h; **F3_code** — скан-код клавиши «F3», которая входит в комбинацию «горячих» клавиш; **key3_code** — скан-код клавиши «3», которая будет блокироваться/разблокироваться при каждом нажатии «горячей» комбинации клавиш; **f** — флаг, который при каждом нажатии «горячей» комбинации клавиш переключается из состояния 0 в 1 или из 1 в 0 (состояние 1 означает, что клавиша «3» заблокирована); **rr** и **sr** — переменные, которые используются для задания значений регистров общего назначения и сегментных регистров соответственно при вызове прерывания.

В главной программе использует символьный массив **string** для проверки работы программы.

Переменные процедуры обработки прерывания 9h:

- **c** — переменная, которая используется для подтверждения приема из клавиатуры, в случае, если была нажата клавиша «3», а флаг **f** показывал, что эта клавиша заблокирована;
- **x**, **y** — переменные, которые используются для сохранения координат курсора на экране в момент вызова процедуры обработки прерывания;
- **byte17** — байт флага состояния клавиатуры в области данных BIOS по адресу 0040:0017;
- **byte18** — байт флага состояния клавиатуры в области данных BIOS по адресу 0040:0018;
- **mask** — маска, которая используется для определения нажатия клавиши левый **Shift** (в этом случае бит 1 в byte17 установлен в 1);
- **mask17** — маска, которая используется для определения нажатия клавиши **Ctrl** (в этом случае бит 2 в byte17 установлен в 1);

- **mask18** — маска, которая используется для определения нажатия клавиши левый **Ctrl** (в этом случае бит 0 в byte18 установлен в 1);

Описание алгоритма программы

Главная программа выполняет такие действия:

- Запоминает адрес старого обработчика прерывания 9h, вызывая функцию readvect(in) с параметром in=9.
- Записывает в таблицу векторов прерываний адрес нового обработчика прерывания с помощью функции writevect().
- Вводом строки символов дает возможность проверить работу программы и ее реакцию на нажатие «горячей» комбинации клавиш и блокирование/разблокирование ввода клавиши «3».
- В конце работы восстанавливает в таблице векторов прерываний адрес старого обработчика.

Для решения задачи процедура обработки прерывания от клавиатуры new9() должна действовать по такому алгоритму:

- Прочитать флаги состояния клавиатуры (статус клавиш-переключателей), которые находятся в области данных BIOS (два байта по адресам 0040:0017 и 0040:0018).
- Выделить бит 1 в флаге по адресу 0040:0017 (если он равен 1, то нажата клавиша левый Shift).
- Выделить бит 2 в этом же флаге (если он равен 1, то нажата левый или правый Ctrl).
- Выделить бит 0 в флаге состояния клавиатуры по адресу 0040:0018 (если он равен 1, то нажата клавиша левый Ctrl).
- Из порта 60h прочитать скан-код нажатой клавиши.
- Если нажата комбинация клавиш левый Shift, правый Ctrl (нажата клавиша Ctrl, но это не правый Ctrl)и клавиша F3, то выполнить п.7. Иначе — перейти к п.8.
- Сигнализировать о нажатии «горячей» комбинации клавиш звуковым сигналом, переключить значение флага блокирования ввода клавиши «3» на обратное и вызвать старый обработчик прерывания от клавиатуры.

- Прочитав байт из порта 60h, определить, нажата ли клавиша «3» и если, кроме этого, еще и флаг блокирования указывает на то, что она заблокирована (f=1), то выполнить п.п. 9 и 10, иначе — вызвать старый обработчик прерывания.
- Послать подтверждение приема в клавиатуру. Для этого в порт 61h на короткое время выставить «1» по шине старшего разряда.
- Сбросить контроллер прерываний, посылая код 20h в порт 20h.

Функция readvect() читает вектор заданного прерывания. Для чтения вектора используется функция 35h DOS (прерывания 21h):

```
Вход: AH = 35h;
AL = номер вектора прерывания.
Выход: ES:BX = адрес программы обработки прерывания
```

Функция **writevect()** устанавливает новый вектор прерывания на заданный адрес. Для записи вектора используется функция 25h DOS:

```
Вход: AL = номер вектора прерывания;
DS:BX = 4-байтный адрес нового обработчика прерывания.
```

Текст программы

```
/*--------Лабораторная работа N5---------------*/
/*---------Управление клавиатурой---------------*/
/* Подключение стандартных заголовков */
#include <dos.h>

void interrupt (*old9)(); /* Старый обработчик прерывания 9h */
void interrupt new9();  /* Новый обработчик прерывания 9h */
void *readvect (int in); /* Чтение вектора         */
void writevect (int in,void *h); /* Запись вектора      */

unsigned char F3_code=61; /* scan-code "F3"  */
unsigned char key3_code=4;/* scan-code "3"   */
char f=0;          /* Флаг           */
union REGS rr;
struct SREGS sr;

/*-----------------------------------------------*/
void main()
{
 char string[80]; /* Буфер для ввода текста */
```

```
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("---------------");
 cprintf(" Лабораторная работа N5      ");
 cprintf("---------------");
 cprintf("---------------");
 cprintf("  Управление клавиатурой      ");
 cprintf("---------------");

 old9=readvect(9);
 writevect(9,new9);
 textattr(0x0c);
 cprintf("\n\n\r\"горячая\" комбинация: ");
 textattr(0x0a);
 cprintf("Left Shift, Right Ctrl, F3\n\r");
 textattr(0x0b);
 cprintf("Клавиша, которая блокируется: ");
 textattr(0x0f);
 cprintf("3");
 textattr(0x07);
 cprintf("\r\nВводите строку символов>");
 scanf("%s",string);
 writevect(9,old9);
}
/*------------------------------------------------*/
/* Чтение вектора */
void *readvect(int in)
{
 rr.h.ah=0x35;
 rr.h.al=in;
 intdosx(&rr,&rr,&sr);
 return(MK_FP(sr.es,rr.x.bx));
}
/*------------------------------------------------*/
/* Запись вектора */
void writevect(int in,void *h)
{
 rr.h.ah=0x25;
 rr.h.al=in;
 sr.ds=FP_SEG(h);
 rr.x.dx=FP_OFF(h);
 intdosx(&rr,&rr,&sr);
```



```
}
/*------------------------------------------------*/
/* Новый обработчик 9-го прерыванмя */
void interrupt new9()
{
 unsigned char c,x,y;
 unsigned char byte17,byte18;
 unsigned char mask=0x02;
 unsigned char mask17=0x04;
 unsigned char mask18=0x01;

 byte17=peekb(0x40,0x17);
 byte18=peekb(0x40,0x18);
 if((inportb(0x60)==F3_code)&&(byte17&mask)&&
            (byte17&mask17)&&(!(byte18&mask18)))
 {
  cputs("\7");
  x=wherex();
  y=wherey();
  gotoxy(55,3);
  textattr(0x1e);
  if(f==0)
  {
  f=1;
  cprintf("Клавиша \"3\" заблокирована ");
  }
  else
  {
  f=0;
  cprintf("Клавиша \"3\" разблокирована");
  }
  gotoxy(x,y);
  textattr(0x07);
  (*old9)();
 }
 if( (f==1) && (inportb(0x60)==key3_code) )
 {
  c=inportb(0x61);
  outportb(0x61,c|0x80);
  outportb(0x61,c);
  outportb(0x20,0x20);
 }
 else
  (*old9)();
}
```

Результаты работы программы

Во время программы при первом нажатии комбинации клавиш **LeftCtrl+RightShift+F3** программа переходит в режим блокирования вводу клавиши 3, при втором — отменяет этот режим.



Лабораторная работа №6. Управление таймером

Цель работы

Изучение функций системного таймера и закрепление практических навыков работы с ним.

Постановка задачи

Построить модель аналого-цифрового преобразователя (АЦП), которая работает в реальном времени. Процесс, который дискретизуется, моделируется программой (программным блоком), который выполняет циклическое вычисление функции **y=F(x)**, где **x** — номер итерации. Преобразователь моделируется программой, которая выполняет с заданной частотой (в реальном времени) прерывание процесса, считывание и запоминание текущего значения функции. Запомнить не меньше 80 значений функции. Обеспечить наглядное представление результатов работы «АЦП».

Индивидуальные задания

Для получения более наглядного представления «процесса» допускается подбирать другие коэффициенты функции. Частоту дискретизации выдерживать с точностью до 1 гц.

Пример решения задачи

Индивидуальное задание

```
функция - y=50*(sin(x/10)+cos(x/8))+R+150;
R - в диапазоне 0 - 10;
частота - 36.4 Гц.
```


Разработка алгоритма решения

Структура программы

Программа состоит из основной программы и трех функций.

- **void *readvect(int in)** — функция читает вектор прерывания с номером in и возвращает его значение.
- **void writevect (int in, void *h)** — функция устанавливает новый вектор прерывания in на новый обработчик этого прерывания по адресу h.
- **void interrupt newtime()** — процедура нового обработчика прерывания таймера.

Описание переменных и констант

В этой программе применяются две константы:

- **TIMEINT=8 —** номер прерывания таймера;
- **NN=100** — максимальное число показаний АЦП.

Переменные, глобальные для всей программы:

- **y** — массив показаний АЦП;
- **ny** — текущий индекс в массиве показаний;
- **yc** — текущее значение функции;
- **kf** — счетчик вызовов **oldtime** (**oldtime** вызывается каждые второй раз);
- **rr** и **sr** — переменные, которые используются для задания значений регистров общего назначения и сегментных регистров, соответственно при вызове прерывания.

Переменные главной программы:

- **oldtic** — старый коэффициент деления;
- **newtic** — новый коэффициент деления (применяется для увеличения частоты вызова прерываний таймера);
- **x** — аргумент заданной функции **F(x)**;
- **dd —** тип графического адаптера;
- **m** — режим графики;
- **errorcode** — код результата инициализации графики.

Описание алгоритма программы

Программу можно назвать моделью процесса АЦП. Главная программа постоянно вычисляет значения заданной функции **F(x)** при переменном аргументе, что имитирует непрерывный сигнал, а обработчик прерывания 8 имитирует преобразователь с постоянным шагом дискретизации по времени. Перед началом работы канал 0 таймера программируется на частоту в 2 рази большую обычной (записью в порт 43h управляющего байта 00110110b=36h, а потом посылкой в порт 40h нового значения коэффициента деления), таким образом, «частота дискретизации» составляет около 36.4 Гц. При поступлении следующего прерывания запоминается текущее значение функции **F(x)**, старый обработчик прерывания oldtime вызывается не при кожному прерывании, а лишь один раз из двух (переменная **kf** — счетчик по модулю 2), когда **oldtime** не вызывается, наш обработчик сам сбрасывает контроллер прерываний посылкой значения 20h в порт 20h. После набора 100 «показаний АЦП» восстанавливается старый вектор обработчика таймера, а результат аналого-цифрового преобразование выводится на терминал в графическом режиме в виде решетчатой функции.

Функция **readvect()** читает вектор заданного прерывания. Для чтения вектора применяется функция 35h DOS (прерывания 21h):

```
Вход: AH = 35h;
AL = номер вектора прерывания.
Выход: ES:BX = адрес программы обработки прерывания.
```

Функция **writevect()** устанавливает новый вектор прерывания по заданному адресу. Для записи вектора применяется функция 25h DOS:

```
Вход: AH = 25h;
AL = номер вектора прерывания;
DS:BX = 4-байтный адрес нового обработчика прерывания.
```

Текст программы

```
/*-------Лабораторная работа N6----------------*/
/*-------"Управление таймером"------------------*/

/* Подключение стандартных заголовков */
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <graphics.h>
#include <time.h>
#include <conio.h>

#define TIMEINT 8   /* Прерывание таймера  */
```



```
#define NN 100 /* Максимальное количество показаний */

void interrupt (*oldtime)(); /* Новый обработчик  прерываний
таймера */

void interrupt newtime(); /* Старый обработчик прерываний таймера
*/
static int y[NN]; /* Накопитель показаний  */
static int ny;   /* Индекс в массиве y  */
static int yc;   /* Текущее значение    */
static int kf; /* Счетчик вызовов oldtime  */
union REGS rr;   /* Запись нового вектора */
struct SREGS sr;
void *readvect(int in);   /* Получение старого вектора */
void writevect(int in, void *h); /* Запись нового вектора */
/*----------------------------------------------*/
void main()
{
 unsigned oldtic=65535u; /* Старый коэфф. деления   */
 unsigned newtic=32768u; /* Новый коэфф. деления    */
 int dd,          /* Графический драйвер */

   m,           /* Графический режим   */
   errorcode;      /* Код ошибки      */
 double x;   /* Аргумент функций sin и cos */

 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("  Лабораторная работа N6        ");
 cprintf("\n   Управление таймером        ");
 textattr(0x8e);
 gotoxy(35,12);
 cprintf("Please wait");
 /* Программирование канала 0 */
 outportb(0x43,0x36);   /* Управляющий байт */
 outportb(0x40,newtic&0x00ff); /* Младший байт счетчика */
 outportb(0x40,newtic>>8);  /* Старший байт счетчика */
 ny=-1; /* Признак того, что АЦП еще не началось */
 kf=15;
 /* Подключение к вектору */
 oldtime=readvect(TIMEINT);
 writevect(TIMEINT,newtime);
```

```
/* Запуск "непрерывного процесса" */
randomize();
for (x=ny=0; ny<NN; x+=1)
 yc=(int)(50*(sin(x/10)+cos(x/8))+random(11)+150);
/* Восстановление вектора */
writevect(TIMEINT,oldtime);
/* Восстановление канала 0 */
outportb(0x43,0x36);    /* Управляющий байт */
outportb(0x40,oldtic&0x00ff); /* Младший байт счетчика */
outportb(0x40,oldtic>>8);  /* Старший байт счетчика */

/* Вывод запомненных результатов */
dd=3; /* EGA, 16 цветов */
m=1; /* Режим 640*350 */
initgraph(&dd,&m,"");
/* проверка результата инициализации */
errorcode = graphresult();
if (errorcode != grOk)  /* ошибка графического режима */
{
 printf("Graphics error: %s\n", grapherrormsg(errorcode));
 printf("Press any key to halt:");
 getch();
 exit(1);      /* аварийное завершение */
}
setcolor(10);
settextstyle(0,0,2);
outtextxy(15,10,"Результати аналого-цифрового преобразования:");

setcolor(9);
rectangle(15,40,624,330);
setcolor(11);
for(ny=0; ny<NN; ny++)
{
 circle(22+ny*6,330-y[ny]*1,2);
 line(22+ny*6,330,22+ny*6,330-y[ny]*1);
}
setcolor(12);
settextstyle(0,0,1);
outtextxy(260,340,"Нажмите любую клавишу ...");
getch();
closegraph();
}
```



```
/* Новый обработчик прериваний таймера */
void interrupt newtime()
{
 if (--kf<0) {
  /* Виклик oldtime - на 2-й раз */
  (*oldtime)();
  kf=1;
  }
 else /* иначе - сброс контроллера */
  outportb(0x20,0x20);
 if ((ny>=0)    /* Если АЦП началось, */
  &&(ny<NN)) /* и NN показаний еще не набрано, */
   y[ny++]=yc;  /* запоминание очередного показания */
}

/* Получение старого вектора */
void *readvect(int in)
{
 rr.h.ah=0x35; rr.h.al=in;
 intdosx(&rr,&rr,&sr);
 return(MK_FP(sr.es,rr.x.bx));
}

/* Запись нового вектора */
void writevect(int in, void *h)
{
 rr.h.ah=0x25;
 rr.h.al=in;
 sr.ds=FP_SEG(h);
 rr.x.dx=FP_OFF(h);
 intdosx(&rr,&rr,&sr);
}
```

Результаты работы программы

Результат работы представляется в графическом режиме в виде решетчатой функции на экране терминала.



Лабораторная работа №7. Управление видеоадаптером

Цель работы

Изучение особенностей функционирования видеосистемы в текстовом режиме и получение практических навыков работы с видеомонитором в этом режиме.

Постановка задачи

Применяя прямую запись в видеопамять получить на экране оригинальный, желательно динамический видеоэффект. Возможны (но не обязательны) такие варианты видеоэффектов:

- «теннисный мячик» — шарик, который летает по экрану и отражается от рамок и границ экраны;
- «сухой лист» — опадание букв с экрана;
- «жук-пожиратель» — фигурка, которая перемещается по экрану по случайной траектории и «съедает» буквы;
- «удав» — то же, что и «жук», но к тому же он увеличивается в размерах, по мере «поедания» букв;

Пример решения задачи

Индивидуальное задание:

- весь экран (80x25 символов) условно делится на прямоугольники размером (10x5 символов).
- текущий прямоугольник инвертирует экран под собой.
- управлять положением текущего прямоугольника на экране можно с помощью клавиш управления курсором.
- при нажатии клавиши «пробел» текущий прямоугольник обменивается содержимым с левым верхним прямоугольником.

- при нажатии клавиши Enter содержимое прямоугольников экрана начинает перемешивается случайным образом между собой до нажатия любой клавиши.
- после этого, используя клавиатуру, можно восстановить начальный экран или выйти из программы (клавиша Esc).

Разработка алгоритм решения

Структура программы

Программа состоит из основной функции **main()** и семи вспомогательных функций.

- **byte GetSym(x1,y1)** — функция читает символ с заданной позиции экрана дисплея.
- **byte GetAtr(x1,y1)** — функция читает атрибут символа с заданной позиции экрана дисплея.
- **void PutSym(x1,y1,sym)** — функция выводит на экран дисплея символ в заданную позицию (x1,y1).
- **void PutAtr(x1,y1,atr)** — функция меняет на экране дисплея атрибут символа в заданной позиции (x1,y1).
- **void Invert(x1,y1)** — функция инвертирует участок на экране размером (10x5), координаты (x1,y1) задают один из участков на экране.
- **void Change(x,y)** — функция обменивает содержимое текущего участка с содержимым левого верхнего участка на экране. Координаты (x,y) задают положение текущего участка.
- **void RandText(void)** — функция псевдослучайным образом перетасовывает все участки на экране.

Описание переменных

Переменные, глобальные для всей программы:

- xk — координата X текущего участка;
- yk — координата Y текущего участка;
- Координаты участка задаются в пределах: X — [0..7], Y — [0..4] .

Описание алгоритма программы

Основная функция **main()** проверяет, был ли в командной строке дополнительный параметр. Если нет, программа не очищает старый экран. Если какой-нибудь параметр был, то экран очищается и выводится инструкция по управлению программой. Далее в основной программе выполняется бесконечный цикл, в котором обрабатываются коды нажатых клавиш и, в зависимости от них, вызываются вспомогательные функции. Выход из цикла — по клавише **Esc**.

Функции **GetSym(x1,y1)**, **GetAtr(x1,y1)** читают непосредственно из видеопамяти дисплея символ и атрибут соответственно.

Функции **PutSym(x1,y1,sym)**, **PutAtr(x1,y1,atr)** выводят непосредственно в видеопамять дисплея символ и атрибут соответственно.

Во всех этих четырех функциях координаты задаются в квадрате 79x24 символов (нумерация начинается с нуля).

Функция **Invert(x1,y1)** использует функции **GetAtr** и **PutAtr** для инверсии прямоугольника экрана размером 10x5 по маске 0x7f, при этом независимо выполняется инверсия фона и цвета символа.

Функция **Change(x,y)** обменивает содержимое текущего участка с содержимым левого верхнего участка путем последовательного побайтного обмену атрибутов и символов. Она использует функции **GetSym**, **GetAtr**, **PutSym**, **PutAtr**.

Функция **RandText(void)** — псевдослучайным образом перетасовывает все участки на экране, при этом она в цикле увеличивает на единицу локальные в данной функции координаты текущего участка **xk**, **yk** и обращается к функции **Change**. Таким образом достигается эффект перемешивания. Функция работает, пока не будет нажата любая клавиша.

Текст программы

```
/*------Лабораторная работа N7-----------------*/
/*-----Управление видеоадаптером.---------------*/
#include <dos.h>

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
/*--------------Константы---------------------- */
#define VSEG 0xb800    /* Сегментный адрес видеопамяти */
#define byte unsigned char
```



```
#define word unsigned int
#define Esc 27
#define Spase 32
#define Enter 13
#define Up 0x48
#define Down 0x50
#define Left 0x4b
#define Right 0x4d
#define Home 0x47
int xk,yk;
/*----Чтение символа из видеопамяти-------------*/
byte GetSym(x1,y1)
int x1,y1;
{
 return(peekb(VSEG,y1*160+x1*2));
}
/*---Чтение атрибута из видеопамяти-------------*/
byte GetAtr(x1,y1)
int x1,y1;
{
 return(peekb(VSEG,y1*160+x1*2+1));
}
/*---Запись символа в видеопамять--------------*/
void PutSym(x1,y1,sym)
int x1,y1;
byte sym;
{
 pokeb(VSEG,y1*160+x1*2,sym);
}
/*---Запись атрибута в видеопамять--------------*/
void PutAtr(x1,y1,atr)
int x1,y1;
byte atr;
{
 pokeb(VSEG,y1*160+x1*2+1,atr);
}
/*--Инверсия квадрата на экране-----------------*/
void Invert(x1,y1)
int x1,y1;
{
 byte b;
 int i,j;
 for (j=0;j<5;j++)
```

```
 for (i=0;i<10;i++)
  {
  b=GetAtr(x1*10+i,y1*5+j);
  PutAtr(x1*10+i,y1*5+j,(b^0x7f));
  }
}
/*--Замена текущего квадрата на левый верхний--*/
void Change(x,y)
byte x,y;
{
 int i,j;
 byte ba,bs;
 if ((x!=0)||(y!=0))
 for (j=0;j<5;j++)
  for (i=0;i<10;i++)
  {
   bs=GetSym(x*10+i,y*5+j);
   ba=GetAtr(x*10+i,y*5+j);
   PutSym(x*10+i,y*5+j,GetSym(i,j));
   PutAtr(x*10+i,y*5+j,GetAtr(i,j));
   PutSym(i,j,bs);
   PutAtr(i,j,ba);
  }
}
/*--Перемешивание квадратов до нажатия клавиши-*/
void RandText(void)
{
 Invert(xk,yk);
 xk=5;
 yk=1;
 while(!kbhit())
 {
 Change(xk,yk);
 xk++;
 if (xk>7) xk=0;
 yk++;
 if (yk>4) yk=0;
 }
 Invert(xk,yk);
}
/*-----Начало основной программы---------------*/
main(int argn,char *argc[])
{
```



```
int i;

xk=0;
yk=0;
if (argn>1){}
else      /* Если параметров нет, вывод инструкции */
{
textattr(10);
clrscr();
cprintf("---------------");
cprintf("   Лабораторная работа N7        ");
cprintf("---------------");
cprintf("---------------");
cprintf("  Управление видеоадаптером.      ");
cprintf("---------------");
textattr(15);
gotoxy(23,4);cprintf("Демонстрация работы с видеопамятью.");
textattr(12);
gotoxy(30,6);cprintf("<< М О З А И К А >>");
textattr(14);
gotoxy(30,8);cprintf("Клавиши управления:");
gotoxy(7,10);cprintf("< Left, Right, Up, Down> - ");
   cprintf("управление выделенним квадратом.");
gotoxy(7,11);cprintf("<Spase Bar> - Обмен содержимым ");
       cprintf("между выделенным квадратом");
gotoxy(7,12);cprintf("      и левым верхним");
       cprintf(" квадратом.");
gotoxy(7,13);cprintf("<Enter> - перемешивание квадратов");
       cprintf(" до нажатия любой клавиши.");
gotoxy(7,14);cprintf("<Esc> - вихід.");
textattr(11);
gotoxy(28,16);cprintf("З А Д А Ч А  И Г Р Ы :");
gotoxy(14,17);cprintf("Собрать при помощи клавиш ");
      cprintf("управления начальный экран.");
textattr(12);
gotoxy(27,19);cprintf("Ж е л а е м  у с п е х а !");
textattr(7);
gotoxy(1,21);cprintf("Примечание: При запуске с ");
       cprintf("параметром <->");
gotoxy(13,22);cprintf("начальным экраном для игры ");
       cprintf("является текущий.");
}
Invert(xk,yk);
```

```
    for(i=0;i==0;)
    switch(getch())
    { /* Обработка нажатых клавиш */
     case Esc: i++; break;
     case Enter:RandText();break;
     case Spase:Invert(xk,yk);

          Change(xk,yk);
          Invert(xk,yk);
          break;
     case 0:
      switch (getch()) {
        case Left:Invert(xk,yk);
             xk--;
             if(xk<0) xk=7;
             Invert(xk,yk);
             break;
        case Right:Invert(xk,yk);
             xk++;
             if(xk>7) xk=0;
             Invert(xk,yk);
             break;
        case  Up:Invert(xk,yk);
             yk--;
             if(yk<0) yk=4;
             Invert(xk,yk);
             break;
        case Down:Invert(xk,yk);
             yk++;
             if(yk>4) yk=0;
             Invert(xk,yk);
             break;
        }
     }
   Invert(xk,yk);
   }
```

Результаты работы программы

Результаты работы программы выводятся на экран терминала и меняются интерактивно.



Лабораторная работа №8. Главная загрузочная запись

Цель работы

Получение практических навыков в работе с Главной Загрузочной Записью жесткого диска.

Постановка задачи

Прочитать и выполнить форматный вывод на экран Главной Загрузочной Записи жесткого диска на своем рабочем месте.

Порядок выполнения

Порядок выполнения работы и содержание отчета определены в общих указаниях.

Пример решения задачи

Разработка алгоритма решения

Программа состоит из основной программы **main()**, которая реализует все действия для чтения Главной Загрузочной Записи.

Описание переменных

Переменные в основной программе:

- x, y — экранные координаты;
- head — номер головки (0);
- Sect_Trk — номер дорожки и сектора (0,1);
- ndrive=0 — номер логического диска;
- EndList — указатель на подпись.

Кроме того, в программе есть такие структуры:

- Структура элемента раздела:

```
struct Part {
 byte ActFlag;   /* признак активного раздела */
```

```
      /* физический адрес начала раздела */
    byte Begin_Hd;          /* # головки */
    word Begin_SecTrk;  /* # сектора та дорожки */
      byte SysCode;      /* код системы */
       /* физический адрес конца раздела */
       byte End_Hd;      /* # головки */
       word End_SecTrk;    /* # сектора и дорожки */
      dword RelSec;  /* # сектора початку */
      dword Size;  /* количество секторов */
      };
```

Структура главной загрузочной записи.

```
struct MBR {
char LoadCode[0x1be]; /* программа загрузки */
struct Part rt[4];  /* 4 элемента разделов */
word EndFlag;           /* подпись MBR */
};
```

Описание алгоритма программы

Эта программа демонстрирует разделение логического диска.

Начальный адрес для чтения задается: 0,0,1. При помощи прерывания 0x13 программа считывает сектор по заданному адресу, далее происходит поэлементный анализ таблицы разделов — пока не встретится признак конца таблицы или раздел нулевого размера. Значения полей элемента таблицы выводятся на экран. Манипуляции, которые описываются макросами TRK и SECT, обеспечивают распаковку номера дорожки и сектора. Если в поле SysCode содержится признак расширенного раздела, то устанавливается новый дисковый адрес, считывается новый сектор и анализируется новая таблица.

Текст программы

```
/*-------Лабораторная работа N8----------------*/
/*---"Главная загрузочная запись"--------------*/
/* Стандартные заголовки */
#include <dos.h>
#include <conio.h>

/* Типы данных */

#define byte unsigned char
#define word unsigned int
#define dword unsigned long
```



```
void read_MBR(void);  /* Чтение MBR */
/* Получение из упакованного SecTrk # сектора */
#define SECT(x) x&0x3f
/* Получение из упакованного SecTrk # дорожки */
#define TRK(x) (x>>8)|((x<<2)&0x300)

 /* структура элемента раздела */
 struct Part {
  byte ActFlag; /* признак активного раздела */
      /* физический адрес начала раздела */
  byte Begin_Hd;          /* # головки */
  word Begin_SecTrk; /* # сектора и дорожки */
  byte SysCode;         /* код системы */
       /* физический адрес конца раздела */
  byte End_Hd;           /* # головки */
  word End_SecTrk;  /* # сектора и дорожки */
  dword RelSec;     /* # сектора початку */
  dword Size;      /* количество секторов */
  };
 /* структура главной загрузочной записи */
 struct MBR {
char LoadCode[0x1be];  /* программа загрузки */
struct Part rt[4];   /* 4 эл-та разделов */
word EndFlag;      /* подпись MBR */
} mbr;
/* дополнительные переменные */
int x=10,y;   /* экранные координаты */
byte head=0;   /* номер головки (0) */
word Sect_Trk=1; /* номер дорожки и сектора (0,1) */
int ndrive=0;  /* номер логического диска */
word *EndList;  /* указатель на подпись */
union REGS rr;
struct SREGS sr;
word i;
/*---------------------------------------------*/
main()
{
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("       Лабораторная работа N8");
 gotoxy(1,2);
 cprintf("      Главная загрузочная запись");
```

```
 textattr(12);
 gotoxy(30,4);
 cprintf("Разделы жесткого диска:\n");
 gotoxy(1,6);
 textattr(11);
 cprintf("Лог.диск -----> \n\r");
 cprintf("Признак ------> \n\r");
 cprintf("Код системы --> \n\r");
 cprintf("Начало: гол.--> \n\r");
 cprintf("    дор.--> \n\r");
 cprintf("    сект.-> \n\r");
 cprintf("Конец: гол.--> \n\r");
 cprintf("    дор. -> \n\r");
 cprintf("    сект.-> \n\r");
 cprintf("Нач.сектор ---> \n\r");
 cprintf("Размер -------> \n\r");
 textcolor(11);
NEXT:
 read_MBR();
 for (EndList=(word *)&mbr.rt[(i=0)];
   (*EndList!=0xaa55)&&(mbr.rt[i].Size>0L);
   EndList=(word *)&mbr.rt[++i])
 {
  /* координаты курсора */
  y=6;
  x+=7;
  gotoxy(x,y++);
  if (mbr.rt[i].SysCode==5)
   {textattr(13);
   cprintf("Ext  ");

   }
  else
   textattr(12);
  cprintf("%-7c",'C'+ndrive++);

  gotoxy(x,y++); textattr(14);
  cprintf("%02xH    ",mbr.rt[i].ActFlag);
  gotoxy(x,y++); textattr(15);
  cprintf("%-7d",mbr.rt[i].SysCode);
  gotoxy(x,y++); textattr(14);
  cprintf("%-7d",mbr.rt[i].Begin_Hd);
  gotoxy(x,y++); textattr(15);
```



```
  cprintf("%-7u",TRK(mbr.rt[i].Begin_SecTrk));
  gotoxy(x,y++); textattr(14);
  cprintf("%-7u",SECT(mbr.rt[i].Begin_SecTrk));
  gotoxy(x,y++); textattr(15);
  cprintf("%-7d",mbr.rt[i].End_Hd);
  gotoxy(x,y++); textattr(14);
  cprintf("%-7u",TRK(mbr.rt[i].End_SecTrk));
  gotoxy(x,y++); textattr(15);
  cprintf("%-7u",SECT(mbr.rt[i].End_SecTrk));
  gotoxy(x,y++); textattr(14);
  cprintf("%-7lu",mbr.rt[i].RelSec);
  gotoxy(x,y++); textattr(15);
  cprintf("%-7lu",mbr.rt[i].Size);
  if (mbr.rt[i].SysCode==5)
  {
   /* если код системы 5, раздел содержит свою таблицу  разделов;
определяется ее дисковый адрес,  и новая таблица считывается в
память */
   head=mbr.rt[i].Begin_Hd;
   Sect_Trk=mbr.rt[i].Begin_SecTrk;
   goto NEXT;
  }
 }
 gotoxy(x,y++);
 textattr(10+128);
 gotoxy(29,18);

 cprintf("Нажмите любую клавишу...");
 getch();
}

/*-----------Читання MBR-----------------------*/
void read_MBR(void)
{
  rr.h.ah=2;      /* Чтение */
  rr.h.al=1;      /* Секторов 1 */
  rr.h.dl=0x80;    /* Жесткий диск */
  rr.h.dh=head;    /* Головка */
  rr.x.cx=Sect_Trk;  /* Дорожка, сектор */
  sr.es=FP_SEG(&mbr); /* Адрес буфера в ОП */
  rr.x.bx=FP_OFF(&mbr);
  int86x(0x13,&rr,&rr,&sr);
  /* Проверка ошибок чтения */
```

```
    if (rr.x.cflag)
    {
     printf("Ошибка чтения: %x. ",rr.h.ah);
     printf("Нажмите любую клавишу...\n\7");
     getch();
     exit();
    }
}
```

Результаты работы программы

В процессе работы программы на экран выводится информация такого вида:

```
        Лабораторная работа N8
       Главная загрузочная запись
Разделы жесткого диска:

Лог.диск -----> C    Ext   E    Ext   G
Признак ------> 80H   00H   00H   00H   00H
Код системы --> 1    5    4    5    0
Начало:  гол.--> 1    0    1    0    1
      дор.--> 0    121   121   724   724
      сект.-> 1    1    1    1    1
Конец:  гол.--> 4    4    4    4    4
     дор. -> 120   975   723   975   975
      сект.-> 17    17    17    17    17
Нач.сектор ---> 17    10285 17    51255 17
Размер -------> 10268 72675 51238 21420 21403

         Нажмите любую клавишу...
```



Лабораторная работа №9. Дисковые структуры данных DOS

Цель работы

Получение практических навыков в работе с Таблицей Размещения Файлов.

Постановка задачи

Определить номера всех кластеров диска, которые занимает заданный преподавателем файл в текущем каталоге.

Пример решения задачи

Разработка алгоритма решения

Программа состоит из главной функции **main()** и одиннадцати вспомогательных функций.

- **void Read_Mbr(void)** — функция чтения MBR и поиска требуемого раздела.
- **void Read_Boot(void)** — функция чтения boot-сектора.
- **void Get_First(void)** — функция определения абсолютного номера сектора начала логического диска.
- **void Read_Fat(void)** — функция чтения FAT.
- **void Read_13(void *mem) —** функция чтения сектора с помощью прерывания 13.
- **void Sect_to_Daddr(dword sect)** — функция формирования физического дискового адреса из номера сектора.
- **dword Clust_to_Sect(word clust)** — функция определения номера сектора по номеру кластера.
- **word Next_Clust(word clust)** — функция выборки следующего кластера из FAT.
- **char *Get_Name(char *s, char *d)** — функция выделения следующего элемента из строки-задания.
- **int Find_Name() —** функция поиска имени в каталоге.
- **void End_of_Job(int n)** — функция выдачи сообщений или аварийного завершения.

Описание переменных

В программе описаны структуры такого вида:

Физический дисковый адрес:

```
struct DADDR {
 byte h; /* головка          */
 word s, /* сектор          */
    t, /* дорожка          */
```

```
    ts; /* сектор, дорожка упакованные */
 };
```

Структура элемента раздела;

```
struct PART {
 byte Boot,        /* признак активного   */
 /* физический адрес начала раздела       */
 Begin_Hd;      /* # головки      */
 word Begin_SecTrk; /* # сектора и дорожки */
 byte SysCode,       /* код системы  */
  /* физический адрес конца раздела  */
 End_Hd;       /* # головки   */
 word End_SecTrk; /* # сектора и дорожки */
 dword RelSec,       /* # сектора початку  */
 Size;        /* количество секторов  */
};
```

Структура Главной Загрузочной Записи:

```
struct MBR
    {
char LoadCode[0x1be]; /* программа загрузки */
struct PART rt[4]; /* 4 элемента разделов  */
word EndFlag;     /* подпись MBR      */
    };
```

Структура загрузочной записи логического диска:

```
struct BootRec {
 byte jmp[3], ident[8];
 word SectSize;
 byte ClustSize;
 word ResSect;
 byte FatCnt;
 word RootSize, TotSecs;
 byte Media;
 word FatSize, TrkSecs, HeadCnt;
 word HidnSecL, HidnSecH;
 dword LongTotSecs;
 byte Drive, reserved1, DOS4_flag;
 dword VolNum; char VolLabel[11], FatForm[8];
 };
```

Структура элемента каталога:

```
struct Dir_Item {
  char fname[11];   /* имя файла      */
  byte attr;      /* атрибут       */
```



```
  byte reserved[10];
  word time;      /* время        */
  word date;      /* дата        */
  word cl;       /* номер 1-го кластера */
  dword size;      /* размер файла    */
  };
```

Переменные, глобальные для всей программы:

- part — текущий элемент раздела;
- buff1[512] — буфер MBR и boot;
- *mbr — указатель на таблицу разделов;
- *boot — указатель на корневую запись;
- buff2[512] — буфер каталога и текста;
- *dir — указатель на часть каталога;
- *text — указатель на текстовый буфер;
- *fat — указатель на FAT;
- job[81] — строка-задание;
- jobptr — текущий указатель в job;
- cname[12] — текущее имя для поиска;
- Fdisk — физический номер диска;
- caddr — текущий дисковый адрес;
- sect — текущий номер сектора;
- clust — текущий номер кластера;
- fat16 — признак формата FAT;
- fsize — размер файла;
- dirnum — номер элемента в каталоге;
- FirstSect — абсолютный номер сектора начала;
- rootdir=1 — признак корневого каталога или подкаталога (1/0);
- lastsect — последний сектор при чтении;
- fatalloc=0 — признак выделения памяти.

Описание алгоритм программы

Функция **main** запрашивает имя файла, потом обрабатывает его и, если все нормально, то запускает вспомогательные функции необходимые для просмотра FAT заданного файла.

Функция **Read_Mbr** выполняет выборку элемента таблицы разделов для заданного диска.

Функция **Read_Boot** считывает boot-сектор логического диска, причем для гибкого диска адрес этого сектора назначается — 0, 0, 1, а для жесткого — выбирается из **part**.

Функция **Get_First** определяет абсолютный номер начального сектора логического диска и сохраняет его переменной **First_Sect**. Это значение вычисляется из физического адреса начала, который берется из полей **Begin_Hd**, **Begin_SecTrk** элемента таблицы разделов.

Функция **Read_Fat** считывает в память FAT целиком, адрес начала FAT на диске и ее размер определяются из ранее прочитанного boot-сектора.

Функция **Read_13** читает один сектор с помощью прерывания BIOS.

Функция **Sect_to_Daddr** преобразует номер логического сектора в физический адрес.

Функция **Clust_to_Sect** преобразует номер кластера в номер сектора.

Функция **Next_Clust** определяет номер следующего кластера, анализируя FAT. Для последнего кластера (и для корневого каталога) эта функция возвращает нулевое значение.

Функция **Get_Name** предназначена для лексического разбора задания, она выделяет из задания очередное слово и переназначает **jobptr**. Пустое (NULL) значение **jobptr** — свидетельство об исчерпании задания.

Функция **Find_Name** выполняет поиск имени в каталоге. Здесь cname — требуемое имя, функция возвращает индекс найденного элемента в массиве **dir** или (-1).

Функция **End_of_Job** выполняет выдачу на экран различных сообщений при ошибках или при завершении программы.

Текст программы

```
/*-------Лабораторная работа N9----------------*/
/*-----"Дисковые структуры данных DOS."--------*/
/* Подключение стандартных заголовков */
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
/*---------------------------------------------*/
/* Типи и структуры данных */
#define byte unsigned char
#define word unsigned int
#define dword unsigned long
#define daddr struct DADDR
struct DADDR { /* физический дисковый адрес */
    byte h;
    word s, t, ts;
    };
struct PART { /* структура элемента раздела */
    byte Boot, Begin_Hd;
    word Begin_SecTrk;
    byte SysCode, End_Hd;
    word End_SecTrk;
    dword RelSec, Size;
    };
struct MBR
 { /* структура Главной Загрузочной Записи */
    char LoadCode[0x1be];
    struct PART rt[4];
    word EndFlag;
    };
struct BootRec
    { /* структура корневой записи */
    byte jmp[3], ident[8];
    word SectSize;
    byte ClustSize;
    word ResSect;
    byte FatCnt;
    word RootSize, TotSecs;
    byte Media;
    word FatSize, TrkSecs, HeadCnt;
```

```
    word HidnSecL, HidnSecH;
    dword LongTotSecs;
    byte Drive, reserved1, DOS4_flag;
    dword VolNum;
    char VolLabel[11], FatForm[8];
   };
struct Dir_Item
 { /* структура элемента директории */
  char fname[11];
  byte attr;
  char reserved[10];
  word time, date, cl;
 dword size;
 };
/*-----------------------------------------------*/

/* Описания функций */
void Read_Mbr(void);
/* Чтение MBR и поиск требуемого раздела       */
void Read_Boot(void); /* Чтение boot-сектора */
void Get_First(void); /* Определение абсолютного номера сектора
начала логического диска    */
void Read_Fat(void);  /* Чтение FAT      */
void Read_13(void *mem);
/* Чтение сектора с омогощью  прерывания 13        */
void Sect_to_Daddr(dword sect);
/* Формирование физического дискового адреса из # сектора */
dword Clust_to_Sect(word clust);
/* Вычисление номера сектора из номера кластера    */
word Next_Clust(word clust);
/* Выборка следующего кластера  из FAT           */
char *Get_Name(char *s, char *d);
/* Выделение следующего элемента из строки-задания   */
int Find_Name(); /* Поиск имени в каталоге */
void End_of_Job(int n);  /* Завершение (при n=0-5 - аварийное)
*/
/*------------------------------------------------*/
/* Переменні */
struct PART part; /* текущий элемент раздела */
byte buff1[512];    /* буфер MBR и boot    */
struct MBR *mbr; /* указатель на таблицу разделов  */
struct BootRec *boot; /* указатель на корневую запись   */
byte buff2[512]; /* буфер каталога и текста  */
```



```
struct Dir_Item *dir; /* указатель на часть каталога  */
char *text;  /* указатель на текстовий буфер */
byte *fat;       /* указатель на FAT  */
char job[81];     /* строка-задание    */
char *jobptr;  /* текущий указатель в job */
char cname[12]; /* текущее имя для поиска  */
byte Fdisk;   /* физический номер диска   */
daddr caddr;  /* текущий дисковый адрес   */
dword sect;      /* текущий номер сектора */
word clust;      /* текущий номер кластера */
byte fat16;      /* признак формату FAT  */
dword fsize;      /* размер файла      */
int dirnum;   /* номер элемента в каталоге  */
dword FirstSect;    /* абс.сектор начала   */
byte rootdir=1; /* признак корневого каталога
              или подкаталога (1/0) */
word lastsect; /* последний сектор при чтении /
byte fatalloc=0; /* признак выделения памяти */
/*-----------------------------------------------*/
main() {
 int n,i;
 textattr(14);
 clrscr();
 /* ввод имени файла */
 cprintf(" Просмотр таблицы FAT. ");
 cprintf("Укажите полное имя файла -->");
 scanf("%s",job);
 /* перевод в верхний регистр */
 strupr(job);
/* проверка правильности идентификатора диска */
 if ((!isalpha(job[0]))||(job[1]!=':')||(job[2]!='\\')) {
  printf("%c%c%c -",job[0],job[1],job[2]);
  End_of_Job(0);
 }
 textattr(10);
 clrscr();
 printf("     Лабораторная работа N9");
 printf("    Дисковые структуры данных DOS.");
 textattr(14);
 cprintf("Файл %s в FAT занимает такие кластеры :\n",job);
 jobptr=job+3;
 if (job[0]>'A') {
  /* для жесткого диска - физический номер и чтение MBR */
```

```
    Fdisk=0x80;
    Read_Mbr();
   }
  else /* для гибкого диска - физический номер */
    Fdisk=job[0]-'A';
   Read_Boot(); /* чтение boot-сектора */
   Read_Fat(); /* чтение FAT */
   dir=(struct Dir_Item *)buff2;
   do { /* рух по каталогам */
    if (!rootdir) clust=dir[dirnum].cl; /* начальный кластер */
  /* выделение следующего элемента из строки-задания */
    jobptr=Get_Name(jobptr,cname);
    do { /* пока не дойдем до последнего кластера */
     if (rootdir) { /* корневой каталог */
  /* нач.сектор корневого кат. и количество секторов */
   sect=boot->ResSect+boot->FatSize*boot->FatCnt;
   lastsect=boot->RootSize*32/boot->SectSize+sect;
     }
     else { /* подкаталог */
      sect=Clust_to_Sect(clust);
      lastsect=boot->ClustSize+sect;
     }
   /* посекторное чтение всего корневого каталога
     или одного кластера подкаталога */
     for (; sect<lastsect; sect++) {
      Sect_to_Daddr(sect);
      Read_13(dir);
      /* поиск имени в прочитанном секторе */
      if ((dirnum=Find_Name())>=0) goto FIND;
     }
     /* до последнего кластера подкаталога */
    }
    while (clust=Next_Clust(clust));
    /* весь каталог просмотрен, а имя не найдено - ошибка */
    printf("%s -",cname);
    if (jobptr==NULL) End_of_Job(4);
    else End_of_Job(5);

  FIND: /* имя найдено */
    rootdir=0;
   }
   while (jobptr!=NULL);
   /* найдено имя файла */
```



```
   /* из каталога получеем 1-й кластер */
   clust=dir[dirnum].cl;
   textattr(7);
   gotoxy(10,4);
   cprintf("Нажимайте любую клавишу ");
   cprintf(" пока не появится <КОНЕЦ ФАЙЛА>.");
   textattr(12);
   gotoxy(1,5);
   cprintf("-<НАЧАЛО ФАЙЛА>");
   gotoxy(1,6);
   cprintf("L->");
   i=0;
   do {
    i++;
    if((i%10)==0) getch();
    textattr(14+16);
    cprintf("%4x",clust);
    textattr(2);
    cprintf("--->");
   }
   while (clust=Next_Clust(clust));
   textattr(12);
   cprintf("<КОНЕЦ ФАЙЛА>\n");
   gotoxy(1,wherey());
   textattr(15+3*16);
   cprintf("Количество кластеров в файле: %u ",i);
   End_of_Job(7);
  }
  /*-----------------------------------------------*/
  /* Чтение MBR и поиск нужного раздела */
  void Read_Mbr(void) {
   int i;
   char ndrive;
   word *EndList;
   caddr.h=0;
   caddr.ts=1;
   ndrive='C';
   mbr=(struct MBR *)buff1;

  NEXT: Read_13(buff1);
   for (EndList=(word *)&mbr->rt[(i=0)];
     (*EndList!=0xaa55)&&(mbr->rt[i].Size>0L);
     EndList=(word *)&mbr->rt[++i]) {
```

```
   if (mbr->rt[i].SysCode==5) {
    caddr.h=mbr->rt[i].Begin_Hd;
    caddr.ts=mbr->rt[i].Begin_SecTrk;
    goto NEXT;
   }
   if (ndrive==job[0]) {
    movmem(&mbr->rt[i],&part,sizeof(struct PART));
    return;
   }
   else ndrive++;
  }
  /* требуемый раздел не найден */
  printf("%c: -",job[0]);
  End_of_Job(1);
 }
 /*----------------------------------------------*/
 /* Чтение boot-сектора */
 void Read_Boot(void) {
  if (Fdisk<0x80) {
   caddr.h=0;
   caddr.ts=1;
  }
  else {
   caddr.h=part.Begin_Hd;
   caddr.ts=part.Begin_SecTrk;
  }
  Read_13(buff1);
  boot=(struct BootRec *)buff1;
  Get_First();
 }
 /*-----------------------------------------------*/
 /* Чтение FAT */
 void Read_Fat(void) {
 dword s, ls;
 byte *f;
  fat=(byte *)malloc(boot->FatSize*boot->SectSize);
  if (fat==NULL) {
   printf("Размещение FAT -");
   End_of_Job(3);
  }
  fatalloc=1;
  s=boot->ResSect;
  ls=s+boot->FatSize;
```



```
  for (f=fat; s<ls; s++) {
   Sect_to_Daddr(s);
   Read_13(f);
   f+=boot->SectSize;
  }
  /* установление формата FAT */
  if (Fdisk>=0x80)
   if (part.SysCode==1) fat16=0;
   else fat16=1;
  else fat16=0;
 }
 /*----------------------------------------------*/
 /* Чтение сектора при помощи прерывания 13 */
 void Read_13(void *mem) {
 /* mem - адреса в ОП */
 union REGS rr;
 struct SREGS sr;
  rr.h.ah=2;
  rr.h.al=1;
  rr.h.dl=Fdisk;
  rr.h.dh=caddr.h;
  rr.x.cx=caddr.ts;
  sr.es=FP_SEG(mem);
  rr.x.bx=FP_OFF(mem);
  int86x(0x13,&rr,&rr,&sr);
  /* Проверка ошибок чтения */
  if (rr.x.cflag&1) {
   printf("%u -",rr.h.ah);
   End_of_Job(2);
  }
 }
 /*-----------------------------------------------*/
 /* Определение абс.номера сектора начала лог.диска */
 void Get_First(void) {
  word s, t;
  if (Fdisk<0x80) FirstSect=0;
  else {
   /* формирование # сектора из физич. дискового адреса */
   t=(part.Begin_SecTrk>>8)|((part.Begin_SecTrk<<2)&0x300);
   s=part.Begin_SecTrk&0x3f;
   FirstSect=(((dword)t*boot->HeadCnt)+part.Begin_Hd)*
   boot->TrkSecs+s-1;
  }
```

```
}
/*-----------------------------------------------*/
/* Формирование физического дискового адреса из # сектора */
void Sect_to_Daddr(dword sect) {
/* sect - номер сектора, caddr - адрес на диске */
dword s;
 if (Fdisk>=0x80) sect+=FirstSect;
 caddr.s=sect%boot->TrkSecs+1;
 s=sect/boot->TrkSecs;
 caddr.h=s%boot->HeadCnt;
 caddr.t=s/boot->HeadCnt;
 caddr.ts=(caddr.t<<8)|caddr.s|((caddr.t&0x300)>>2);
}
/*-----------------------------------------------*/
/* Вычисление номера сектора из номера кластера */
dword Clust_to_Sect(word clust) {
/* clust - номер кластера, возвращает номер сектора */
 dword ds, s;
 ds=boot->ResSect+boot->FatSize*boot->FatCnt+
 boot->RootSize*32/boot->SectSize;
 s=ds+(clust-2)*boot->ClustSize;
 return(s);
}
/*-----------------------------------------------*/
/* Выборка следующего кластера из FAT */
word Next_Clust(word clust) {
/* clust - номер кластера, возвращает номер следующего кластера
  или 0 - если следующего нет */
 word m, s;
 if (rootdir) return(0);
 if (!fat16) {
  m=(clust*3)/2;
  s=*(word *)(fat+m);
  if(clust%2)   /* нечетный элемент */
   s>>=4;
  else  /* четный элемент */
   s=s&0x0fff;
  if (s>0x0fef) return(0);
  else return(s);
 }
 else {
  m=clust*2;
  s=*(word *)(fat+m);
```



```
  if (s>0xffef) return(0);
  else return(s);
 }
}
/*-----------------------------------------------*/
/* Выделение следующего элемента из строки-задания */
char *Get_Name(char *s, char *d) {
/* s - строка задания, d - выделенный элемент, возвращает
указатель на новое начало строки задания. */
char *p,*r;
int i;
 for(i=0;i<11;d[i++]=' ');
 d[11]='\0';
 if ((p=strchr(s,'\\'))==NULL) {
  /* последний элемент строки - имя файла */
  /* перезапись имени */
  for(r=s,i=0; (i<8)&&*r&&(*r!='.'); i++,r++) *(d+i)=*r;
  /* перезапись расширения */
  if (*r) for(i=0,r++; (i<3)&&*r; i++,r++) *(d+8+i)=*r;
  return(NULL);
 }
 else {
  /* следующий элемент - имя подкаталога */
  *p='\0';
  for(r=s,i=0; (i<11)&&*r; i++,r++) *(d+i)=*r;
  return(p+1);
 }
}
/*-----------------------------------------------*/
/* Поиск имени в каталоге */
int Find_Name() {
 int j;

/* cname - найденное имя; возвращает индекс найденного
  элемента в массиве dir или (-1) */
  for (j=0; j<boot->SectSize/sizeof(struct Dir_Item); j++) {
   if (dir[j].fname[0]=='\0') {
  /* конец использованных элементов каталога */
    printf("%s -",cname);
    if (jobptr==NULL) End_of_Job(4);
    else End_of_Job(5);
   }
   if ((byte)dir[j].fname[0]!=0xe5) {
```

```
     if (memcmp(dir[j].fname,cname,11)==0) {
      /* если ім`я збігатся, то:
- при поиске файла элемент не должен иметь
атрибутов "подкаталог" или "метка тома",
- при поиске подкаталога элемент должен иметь атрибут
"подкаталог" */
       if (jobptr==NULL)
        if ( !(dir[j].attr&0x18) ) return(j);
       else
        if (dir[j].attr&0x10) return(j);
     }
    }
   }
 return(-1);
}
/*----------------------------------------------*/
/* Завершение (при n=0-5 - аварийное) */
void End_of_Job(int n) {
/* n - номер сообщения */
static char *msg[] = {
    "неправильный идентификатор диска",
    "логический диск отсутствует",
    "ошибка чтения",
    "нехватка памяти",
    "подкаталог не найден",
    "файл не найден",
    "непредусмотренный конец файла",
    "" };
 /* освобождение памяти */
 if (fatalloc) free(fat);
 /* выдача сообщения */
 textattr(12+128);
 cprintf(" %s\n",msg[n]);
 gotoxy(28,wherey());
 cprintf(" Нажмите любую клавишу...\n");
 textattr(7);
 getch();
 /* завершение программы */
 exit(0);
}
```


Результаты работы программы

В процессе работы программы на экран выводится информация наподобие следующей:

```
Лабораторная работа N9
Дисковые структуры данных DOS.
Файл D:\TC\TC.EXE в FAT занимает такие кластеры:
Нажимайте любую клавишу пока не появится <КОНЕЦ ФАЙЛА>.
-<НАЧАЛО ФАЙЛА>
8L->2410--->2411--->2412--->2413--->2414--->2415--->2416--->2417-
-->2418--->2419--->241a--->241b--->241c--->241d--->241e--->241f-
-->2420--->2421--->2422--->2423--->2424--->2425--->2426--->2427-
-->2428--->2429--->242a--->242b--->242c--->242d--->242e--->242f-
-->2430--->2431--->2432--->2433--->2434--->2435--->2436--->2437-
-->2438--->2439--->243a--->243b--->243c--->243d--->243e--->243f-
-->2440--->2441--->2442--->2443--->2444--->2445--->2446--->2447-
-->2448--->2449--->244a--->244b--->244c--->244d--->244e--->244f-
-->2450--->2451--->2452--->2453--->2454--->2455--->2456--->2457-
-->2458--->2459--->245a--->245b--->245c--->245d--->245e--->245f-
-->2460--->2461--->2462--->2463--->2464--->2465--->2466--->2467-
-->2468--->2469--->246a--->246b--->246c--->246d--->246e--->246f-
-->2470--->2471--->2472--->2473--->2474--->2475--->2476--->2477-
-->2478--->2479--->247a--->247b--->247c--->247d--->247e--->247f-
-->2480--->2481--->2482--->2483--->2484--->2485--->2486--->2487-
-->2488--->2489--->248a--->248b--->248c--->248d--->248e--->248f-
-->2490--->2491--->2492--->2493--->2494--->2495--->2496--->2497-
-->2498--->2499--->249a--->249b--->249c--->249d--->&lt;КОНЕЦ ФАЙЛА>
      Количество кластеров в файле: 142
      Нажмите любую клавишу...
```



Лабораторная работа N10. Управление программами

Цель работы

Изучение принципов управления программами в MS DOS и приобретение практических навыков работы с префиксом программного сегмента и его полями.

Постановка задачи

Разработать программу, производящую форматный вывод на печать своего Префикса Программного Сегмента.

Пример решения задачи

Структура программы

Программа состоит из основной программы и двух функций:

- **void get_DOS_version_h(void)** — функция, возвращающая в глобальной переменной dos_ver старшее число номера версии DOS.
- **void addr_PSP (void)** — функция, получающая сегментный адрес префикса программного сегмента программы и возвращающая его в глобальной переменной pid.

Описание переменных

Переменные, глобальные для всей программы:

- **p_psp** — указатель на структуру struct PSP,
- **pid** — сегментный адрес PSP;
- **dos_ver** — старшее число номера версии DOS;
- **i —** вспомогательная переменная, используемая для просмотра таблицы файлов задачи (JFT), которая представляет собой массив из 20 элементов (хотя возможно, что их число отлично от 20, поэтому размер массива определим из поля JFT_size);
- **l** — переменная, используемая для вывода содержимого сегмента окружения DOS и определения числа строк вызова (для версии DOS 3.0 и выше);
- **s** — переменная, которая вначале используется как указатель на таблицу файлов задачи, затем на строки сегмента окружения и строки вызова;
- **rr** — переменная, которая используется для задания значений регистров общего назначения при вызове прерывания.

Описание алгоритма программы

Данная программа производит распечатку основных полей своего PSP. Для этого префикс программного сегмента представим в виде следующей структуры:


```
struct psp
{    /* ФОРМАТ PSP */
byte ret_op[2];     /* команда INT 20h    */
word end_of_mem;  /* вершина доступной памяти*/
byte reserved1;
byte old_call_dos[5]; /* старый вызов DOS  */
void *term_ptr; /* адрес завершения      */
void *ctrlbrk_ptr;  /* адрес обработчика Ctrl+Break  */
void *criterr_ptr;  /* адрес обработчика крит.ошибок */
word father_psp;   /* PID родителя      */
byte JFT[20];  /* таблица файлов программы */
word env_seg;      /* адрес окружения     */
void *stack_ptr;   /* адрес стека        */
word JFT_size;     /* размер таблицы файлов */
byte *JFT_ptr;     /* адрес таблицы файлов  */
byte reserved2[24];
byte new_call_dos[3]; /* новый вызов DOS    */
} *p_psp;
```

Поле ret_op используется для возможного завершения программы по команде RET 0, поле old_call_dos, содержит команду вызова диспетчера функций DOS. Обращение к этому полю в программе может использоваться вместо команды INT 21h, но в современных версиях DOS для этих целей лучше обращаться к полю new_call _dos.

Поле end_of_mem содержит сегментный адрес конца доступной памяти в системе. В три поля: term_ptr, ctrlbrk_ptr, criterr_ptr DOS при загрузке программы копирует содержимое векторов прерываний: 22h, 23h, 24, представляющее собой адреса обработчиков: завершения программы, комбинации клавиш Ctrl+Break, критической ошибки — соответственно. Предполагается, что программа может свободно перенаправить эти векторы на собственные обработчики соответствующих ситуаций, но от забот по восстановлению векторов программа избавляется, так как при ее завершении DOS сама восстанавливает векторы из соответствующих полей PSP завершаемой программы. Для аналогичных целей предназначено и поле stack_ptr — в нем сохраняется (а при завершении — из него восстанавливается) адрес стека, использовавшегося до вызова программы. Поле, именуемое father_psp, содержит сегментный адрес PSP родителя — программы, запустившей данную программу, обычно родителем является COMMAND.COM.

При загрузке программы DOS, кроме программного сегмента, создает для нее еще и сегмент окружения. Сегмент окружения содержит ASCIIZ-строки, задающие значения некоторых глобальных перемен-

ных, эти значения могут устанавливаться командой DOS SET, они доступны командным файлам и — через PSP — программам. Набор строк окружения заканчивается пустой ASCIIZ-строкой (нулем). В DOS 3.0 и выше за ним следует еще 2-байтное число строк вызова (обычно 1) и далее — строка (или строки) вызова программы. Обычно в первую (до строк вызова) часть порождаемой программы копируется содержимое окружения программы-родителя. Программа имеет доступ к своему сегменту окружения через поле env_seg PSP, содержащее сегментный адрес окружения.

Поле JFT (Job File Table — Таблица Файлов Задачи) представляет собой массив из 20 однобайтных элементов. При открытии программой файла DOS формирует для него блок-описатль в системной таблице файлов и помещает ссылку на него (его номер) в свободный элемент JFT. Дескриптор файла, возвращаемый программе DOS при открытии файла, является номером элемента в JFT. При запуске программы первые пять элементов создаваемой для нее JFT содержат ссылки на системные файлы, остальные свободны. При обработке JFT DOS использует не прямое обращение к полю JFT PSP, а косвенное — через поле JFT_ptr, а в качестве ограничителя размера JFT — не константу 20, а значение поля JFT_size PSP.

После всего, сказанного выше, не составляет труда написать программу, осуществляющую форматный вывод своего префикса программного сегмента. Для в начале необходимо определить версию DOS (с помощью функции get_DOS_version_h() и получить адрес PSP (с помощью функции addr_PSP()).

Функция get_DOS_version_h() определяет старшее число номера версии DOS, используя для этого функцию DOS 30h (прерывание 21h), которая возвращает в регистре AL старшее число номера версии, а в регистре AH — младшее число. Нас интересует только значение регистра AL.

Функция addr_PSP() возвращает сегментный адрес PSP путем использования функции DOS 62h:

```
Вход: AH = 62h
Выход: BX = сегментный адрес PSP текущего процесса
```

Текст программы

```
/*-----Лабораторная работа N10-----------------*/
/*------"Управление программами"----------------*/
/* Подключение стандартных заголовков */
#include <dos.h>
#include <conio.h>
```



```
/* Типы данных */
#define byte unsigned char
#define word unsigned int

/* Описание функций */
void get_DOS_version_h(void); /* Определение версии DOS */
void addr_PSP (void);     /* Получение адреса PSP  */

struct psp
{    /* ФОРМАТ PSP */
byte ret_op[2];    /* команда INT 20h   */
word end_of_mem; /* вершина доступной памяти */
 byte reserved1;
 byte old_call_dos[5]; /* старый вызов DOS  */
 void *term_ptr;    /* адрес завершения   */
 void *ctrlbrk_ptr;  /* адрес обработчика Ctrl+Break  */
 void *criterr_ptr;  /* адрес обработчика крит.ошибок */
 word father_psp;   /* PID родителя      */
 byte JFT[20]; /* таблица файлов программы */
 word env_seg;     /* адрес окружения    */
 void *stack_ptr;   /* адрес стека      */
 word JFT_size; /* размер таблицы файлов  */
 byte *JFT_ptr;   /* адрес таблицы файлов  */
 byte reserved2[24];
 byte new_call_dos[3]; /* новый вызов DOS   */
 } *p_psp;

word pid;        /* сегм.адрес PSP    */
int dos_ver,      /* версия DOS       */
  i, l, j;
char *s;
union REGS rr;

main()
{
 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("---------------");
 cprintf("   Лабораторная работа N10     ");
 cprintf("---------------");
 cprintf("---------------");
 cprintf("       Управление программами  ");
```

```
cprintf("---------------");
textcolor(11);
get_DOS_version_h();
addr_PSP();
/* распечатка PSP */
cprintf("\n\n      Адрес PID = %04X\n\n\r",pid);
p_psp=(struct psp *)MK_FP(pid,0);
textcolor(10);
cprintf("Команды:\n\r");
cprintf("--------\n\r");
textcolor(14);
cprintf("   Завершение - int 20h:");
textcolor(12);
cprintf(" %02X %02X\n\r",p_psp->ret_op[0],p_psp->ret_op[1]);
textcolor(14);
cprintf("   Старый вызов DOS:   ");
textcolor(12);
for (i=0;i<5;cprintf("%02X ",p_psp->old_call_dos[i++]));
textcolor(14);
cprintf("\n\r   Новый вызов DOS:     ");
textcolor(12);
for(i=0;i<3;cprintf("%02X ",p_psp->new_call_dos[i++]));
textcolor(10);
cprintf("\n\n\rАдреса:\n\r");
cprintf("-------\n\r");
textcolor(14);
cprintf("  Конец памяти:        ");
textcolor(12);
cprintf("%04X:0000\n\r",p_psp->end_of_mem);
textcolor(14);
cprintf("  Обработчик завершения:  ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->term_ptr);
textcolor(14);
cprintf("  Обработчик Ctrl+Break:  ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->ctrlbrk_ptr);
textcolor(14);
cprintf("  Обработчик критич.ошибки: ");
textcolor(12);
cprintf("%Fp\n\r",p_psp->criterr_ptr);
textcolor(14);
cprintf("  Стек:            ");
```



```
textcolor(12);
cprintf("%Fp\n\n\r",p_psp->stack_ptr);
textcolor(14);
cprintf("\n\rРодитель: ");
textcolor(12);
cprintf("%04X ",p_psp->father_psp);
textcolor(0x8b);
cprintf("\n\n\rНажмите любую клавишу ...\n\r\7");
getch();
clrscr();
textattr(0x0a);
cprintf("---------------");
cprintf("       Лабораторная работа N10        ");
cprintf("---------------");
cprintf("---------------");
cprintf("       Управление программами        ");
cprintf("---------------");
/* Распечатка таблицы файлов */
s=p_psp->JFT_ptr;
textcolor(10);
cprintf("\n\n\rТаблица файлов: ");
textcolor(12);
cprintf("%Fp (%d) ",s,p_psp->JFT_size);
textcolor(11);
if (s==(byte *)p_psp+0x18)
 cprintf(" - в этом же PSP");
cprintf("\n\r");
for (i=0; ++i<=p_psp->JFT_size; cprintf("%d ",*(s++)));
textcolor(10);
cprintf("\n\n\rОкружение DOS: ");
textcolor(12);
cprintf("%04X\n\r",p_psp->env_seg);
s=(char *)MK_FP(p_psp->env_seg,0);
textcolor(11);
while(l=strlen(s))
{
 cprintf("   %s\n\r",s);
 s+=l+1;

}
if (dos_ver>2)
{
 /* для DOS 3.0 и дальше можно получить строку вызова */
```

```
  s++;
  l=*((int *)s);
  textcolor(10);
  cprintf("\n\rЧисло строк вызова: ");
  textcolor(12);
  cprintf("%d\n\r",l);
  s+=2;
  textcolor(11);
  for(i=0; i<l; i++)
  {
   cprintf("%s\n\r",s);
   s+=strlen(s)+1;
  }
 }
 textattr(0x8b);
 cprintf("\n\n\n\n\rНажмите любую клавишу ...\7");
 textattr(0x07);
 cprintf("\n\r");
 getch();
 clrscr();
}

/* Определение версии DOS */
void get_DOS_version_h(void)
{
 rr.h.ah=0x30;
 intdos(&rr,&rr);
 dos_ver=rr.h.al;
}

/* Получение адреса PSP */
void addr_PSP (void)
{
 rr.h.ah=0x62;
 intdos(&rr,&rr);
 pid=rr.x.bx;
}
```


Результаты работы программы

В процессе работы программы на экран была выведена следующая информация:

```
--------------------------------------------------
--       Лабораторная работа N10       ------
Управление программами       ----
      Адрес PID = 0BA0

Команды:
--------
   Завершение - int 20h: CD 20
   Старый вызов DOS:   9A F0 FE 1D F0
   Новый вызов DOS:    CD 21 CB

Адреса:
-------
  Конец памяти:        9FC0:0000
  Обработчик завершения:  0AFA:02B1
  Обработчик Ctrl+Break:  0AFA:014A
  Обработчик критич.ошибки: 0AFA:0155
  Стек:            0E04:0F94

Родитель: 0AFA

Таблица файлов: 0BA0:0018 (20) - в этом же PSP
1 1 1 0 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

Окружение DOS: 0A1E
   CONFIG=STD
   COMSPEC=C:\DOS\COMMAND.COM
   PROMPT=$p$g
   PATH=D:\WIN;C:\;C:\DOS;C:\ARH;C:\NC;C:\BAT;D:\TP;
D:\TP7;D:\BC\BIN
   TEMP=d:\~TMP

Число строк вызова: 1
D:\TC\TC_LAB10.EXE
```

Что нужно знать для экзамена



Функции прерывания DOS INT 21H

Ниже приведены базовые функции для прерывания DOS INT 21H. Код функции устанавливается в регистре AH:

00

Завершение программы (аналогично INT 20H).

01

Ввод символа с клавиатуры с эхом на экран.

02

Вывод символа на экран.

03

Ввод символа из асинхронного коммуникационного канала.

04

Вывод символа на асинхронный коммуникационный канал.

05

Вывод символа на печать (гл.19).

06

Прямой ввод с клавиатуры и вывод на экран.

07

Ввод с клавиатуры без эха и без проверки Ctrl/Break.

08

Ввод с клавиатуры без эха с проверкой Ctrl/Break.

09

Вывод строки символов на экран.

0A

Ввод с клавиатуры с буферизацией.

0B

Проверка наличия ввода с клавиатуры.

0C

Очистка буфера ввода с клавиатуры и запрос на ввод.

0D

Сброс диска.

0E

Установка текущего дисковода.

0F

Открытие файла через FCB.

10

Закрытие файла через FCB.

11

Начальный поиск файла по шаблону.

12

Поиск следующего файла по шаблону.

13

Удаление файла с диска.

14

Последовательное чтение файла.

15

Последовательная запись файла.

16

Создание файла.

17

Переименование файла.

18

Внутренняя операция DOS.

19

Определение текущего дисковода.

1A

Установка области передачи данных (DTA).

1B

Получение таблицы FAT для текущего дисковода.

1C

Получение FAT для любого дисковода.

21

Чтение с диска с прямым доступом.

22

Запись на диск с прямым доступом.

23

Определение размера файла.

24

Установка номера записи для прямого доступа.

25

Установка вектора прерывания.

26

Создание программного сегмента.

27

Чтение блока записей с прямым доступом.

28

Запись блока с прямым доступом.

29

Преобразование имени файла во внутренние параметры.

2A

Получение даты (CX-год,DH-месяц,DL-день).

2B

Установка даты.

2C

Получение времени (CH-час,CL-мин,DH-с,DL-1/100с).

2D

Установка времени.

2E

Установка/отмена верификации записи на диск.

2F

Получение адреса DTA в регистровой паре ES:BX.

30

Получение номера версии DOS в регистре AX.

31

Завершение программы, после которого она остается резидентной в памяти.

33

Проверка Ctrl/Break.

35

Получение вектора прерывания (адреса подпрограммы).

36

Получение размера свободного пространства на диске.

38

Получение государственно зависимых форматов.

39

Создание подкаталога (команда MKDIR).

3A

Удаление подкаталога (команда RMDIR).

3B

Установка текущего каталога (команда CHDIR).

3C

Создание файла без использования FCB.

3D

Открытие файла без использования FCB.

3E

Закрытие файла без использования FCB.

3F

Чтение из файла или ввод с устройства.

40

Запись в файл или вывод на устройство.

41

Удаление файла из каталога.

42

Установка позиции для последовательного доступа.

43

Изменение атрибутов файла.

44

Управление вводом-выводом для различных устройств.

45

Дублирование файлового номера.

46

«Склеивание» дублированных файловых номеров.

47

Получение текущего каталога.

48

Выделение памяти из свободного пространства.

49

Освобождений выделенной памяти.

4A

Изменение длины блока выделенной памяти.

4B

Загрузка/выполнение программы (подпроцесса).

4C

Завершение подпроцесса с возвратом управления.

4D

Получение кода завершения подпроцесса.

4E

Начальный поиск файла по шаблону.

4F

Поиск следующего файла по шаблону.

54

Получение состояния верификации.

56

Переименование файла.

57

Получение/установка даты и времени изменения файла.

59

Получение расширенного кода ошибки.

5A

Создание временного файла.

5B

Создание нового файла.

5C

Блокирование/разблокирование доступа к файлу.

62

Получение адреса префикса программного сегмента (PSP).



Порты

Порт представляет собой устройство, которое соединяет процессор с внешним миром. Через порт процессор получает сигналы с устройств ввода и посылает сигналы на устройство вывода. Теоретически процессор может управлять до 65 536 портами, начиная с нулевого порта. Для управления вводом-выводом непосредственно на уровне порта используются команды IN и OUT:

- Команда IN передает данные из входного порта в регистр AL (байт) или в регистр AX (слово). Формат команды:

  ```
  IN регистр,порт
  ```

- Команда OUT передает данные в порт из регистра AL (байт) или из регистра AX (слово). Формат команды:

  ```
  OUT порт,регистр
  ```

Номер порта можно указывать статически или динамически:

1. Статическое указание порта возможно при непосредственном использовании значения от 0 до 255:

```
Ввод: IN AL.порт# ;Ввод одного байта
Вывод: OUT порт#,AX ;Вывод одного слова
```

2. Динамическое указание порта устанавливается в регистре DX от 0 до 65535. Этот метод удобен для последовательной обработки нескольких портов. Значение в регистре DX в этом случае увеличивается в цикле на 1. Пример ввода байта из порта 60H:

```
MOV DX,60H ;Порт 60H (клавиатура)
IN AL,DX ;Ввод байта
```

Ниже приведен список некоторых портов (номера в шестнадцатеричном представлении):

`21`

Регистры маски прерывании.

`40...42`

Таймер/счетчик

`60`

Ввод с клавиатуры

`61`

Звуковой порт (биты 0 и 1)

`201`

Управление играми

`3B0...3BF`

Монохромный дисплей и параллельный адаптер печати

`3D0...3DF`

Цветной/графический адаптер

`3F0...3F7`

Дисковый контроллер

В случае, если, например, программа запрашивает ввод с клавиатуры, то она выдает команду прерывания INT 16H. В этом случае система устанавливает связь с BIOS, которая с помощью команды IN вводит байт с порта 60H.

На практике рекомендуется пользоваться прерываниями DOS и BIOS.

Однако можно также успешно обойтись без BIOS при работе с портами 21, 40...42, 60 и 201.

Что нужно знать для семинара



Справочник по директивам языка Ассемблера

Индексная адресация памяти

При прямой адресации памяти в одном из операндов команды указывается имя определенной переменной, например для переменной COUNTER:

```
ADD CX,COUNTER
```

Во время выполнения программы процессор локализует указанную переменную в памяти путем объединения величины смещения к этой переменной с адресом сегмента данных.

При индексной адресации памяти ссылка на операнд определяется через базовый или индексный регистр, константы, переменные смещения и простые переменные. Квадратные скобки, определяющие операнды индексной адресации, действуют как знак плюс (+). Для индексном адресации памяти можно использовать:

- базовый регистр BX в виде [BX] вместе с сегментным регистром DS или базовый регистр BP в виде [BP] вместе с сегментным регистром SS. Например, с помощью команды:
  ```
  MOV DX,[BX] ;Базовый регистр
  ```
  в регистр DX пересылается элемент, взятый по относительному адресу в регистре BX и абсолютному адресу сегмента в регистре DS; — индексный регистр DI в виде [DI] или индексный регистр SI в виде [SI], оба вместе с сегментным регистром DS. Например, с помощью команды:

```
MOV AX,[SI] ;Индексный регистр
```

в регистр AX пересылается элемент, взятый по относительному адресу в регистре SI и абсолютному адресу сегмента в регистре DS;

- [константу], содержащую непосредственный номер или имя в квадратных скобках. Например, с помощью команды
```
MOV [BX+SI+4],AX ;База+индекс+константа
```
содержимое регистра AX пересылается по адресу, который вычисляется, как сумма абсолютного адреса в регистре DS, относительного адреса в регистре BX, относительного адреса в регистре SI и константы 4;

- смещение (+ или -) совместно с индексным операндом. Существует небольшое различие при использовании константы и смещения. Например, с помощью команды:
```
MOV DX,8[DI][4] ;Смещение+индекс+константа
```
в регистр DX пересылается элемент, взятый по абсолютному адресу в регистре DS, смещению 8, относительному адресу в регистре DI и константе 4.

Эти операнды можно комбинировать в любой последовательности. Но нельзя использовать одновременно два базовых регистра [BX + BP] или два индексных регистра [DI + SI]. Обычно индексированные адреса используются для локализации элементов данных в таблицах.

Операторы языка ассемблер

Существует три типа ассемблерных операторов: операторы атрибута, операторы, возвращающие значение, и операторы, специфицирующие битовую строку.

Операторы, специфицирующие битовую строку, оператор MASK, счетчик сдвига и оператор WIDTH относятся к директиве RECORD.

Оператор LENGTH

Оператор LENGTH возвращает число элементов, определенных операндом DUP. Например, следующая команда MOV заносит в регистр DX значение 10:

```
TABLEA DW 10 DUP(?) ...
MOV DX,LENGTH TABLEA
```

В случае, если операнд DUP отсутствует, то оператор LENGTH возвращает значение 01.

Оператор OFFSET

Оператор OFFSET возвращает относительный адрес переменной или метки внутри сегмента данных или кода. Оператор имеет следующий формат:

```
OFFSET переменная или метка
```

Например, команда

```
MOV DX,OFFSET TABLEA
```

устанавливает в регистре DX относительный адрес (смещение) поля TABLEA в сегменте данных. (Заметим, что команда LEA выполняет аналогичное действие, но без использования оператора OFFSET.)

Оператор PTR

Оператор PTR используется совместно с атрибутами типа BYTE, WORD или DWORD для локальной отмены определенных типов (DB, DW или DD) или с атрибутами NEAR или FAR для отмены значения дистанции по умолчанию. Формат оператора следующий:

```
тип PTR выражение
```

В поле «тип» указывается новый атрибут, например BYTE. Выражение имеет ссылку на переменную или константу.

Оператор SEG

Оператор SEG возвращает адрес сегмента, в котором расположена указанная переменная или метка. Наиболее подходящим является использование этого оператора в программах, состоящих из нескольких отдельно ассемблируемых сегментов. Формат оператора:

```
SEG переменная или метка
```

Примеры применения оператора SEG в командах MOV:

```
MOV DX,SEG FLOW ;Адрес сегмента данных
MOV DX,SEG A20 ;Адрес сегмента кода
```

Оператор SHORT

Назначение оператора SHORT — модификация атрибута NEAR в команде JMP, если переход не превышает границы +127 и -128 байт:

```
JMP SHORT метка
```

В результате Ассемблер сокращает машинный код операнда от двух до одного байта. Эта возможность оказывается полезной для коротких переходов вперед, так как в этом случае Ассемблер не может сам определить расстояние до адреса перехода и резервирует два байта при отсутствии оператора SHORT.

Оператор SIZE

Оператор SIZE возвращает произведение длины LENGTH и типа TYPE и полезен только при ссылках на переменную с операндом DUP. Формат оператора:

```
SIZE переменная
```

Оператор TYPE

Оператор TYPE возвращает число байтов, соответствующее определению указанной переменной:

Определение	Число байтов
DB	1
DW	2
DD	4
DQ	8
DT	10
STRUC	Число байтов, определённых в STRUC
NEAR	метка FFFF
FAR	метка FFFE

Формат оператора TYPE:

```
TYPE переменная или метка
```

Директивы ассемблера

Директива ASSUME

Назначение директивы ASSUME — установить для ассемблера связь между сегментами и сегментными регистрами CS, DS, ES и SS. Формат директивы:

```
ASSUME сегментный_регистр:имя [, ... ]
```

В директиве указываются имена сегментных регистров, групп (GROUP) и выражений SEG.

Одна директива ASSUME может назначить до четырех сегментных регистров в любой последовательности, например:

```
ASSUME CS:CODESG,DS:DATASG,SS:STACK,ES:DATASG
```

Для отмены любого ранее назначенного в директиве ASSUME сегментного регистра необходимо использовать ключевое слово NOTHING:

```
ASSUME ES:NOTHING
```

В случае, если, например, регистр DS оказался не назначен или отменен ключевым словом NOTHING, то для ссылки к элементу из сегмента данных в командах используется операнд со ссылкой к регистру DS:


```
MOV AX,DS:[BX] ;Использование индексного адреса
MOV AX,DS:FLDW ;Пересылка содержимого поля FLDW
```

Конечно, регистр DS должен содержать правильное значение сегментного адреса.

Директива EXTRN

Назначение директивы EXTRN — информировать Ассемблер о переменных и метках, которые определены в других модулях, но имеют ссылки из данного модуля. Формат директивы:

```
EXTRN имя: тип [, ... ]
```

Директива GROUP

Программа может содержать несколько сегментов одного типа (код, данные, стек). Назначение директивы GROUP — собрать однотипные сегменты под одно имя так, чтобы они поместились в один сегмент объемом 64 Кбайт, формат директивы:

```
имя GROUP имя сегмента [, ... ]
```

Директива INCLUDE

Отдельные фрагменты ассемблерного кода или макрокоманды могут использоваться в различных программах. Для этого такие фрагменты и макрокоманды записываются в отдельные дисковые файлы, доступные для использования из любых программ. Пусть некоторая подпрограмма, преобразующая ASCII-код в двоичное представление, записана на диске C в файле по имени CONVERT.LIB. Для доступа к этому файлу необходимо указать директиву

```
INCLUDE C:CONVERT.LIB
```

причем в том месте исходной программы, где должна быть закодирована подпрограмма преобразования ASCII-кода. В результате Ассемблер найдет необходимый файл на диске и вставит его содержимое в исходную программу. (В случае, если файл не будет найден, то Ассемблер выдаст соответствующее сообщение об ошибке и директива INCLUDE будет игнорирована.) Для каждой вставленной строки Ассемблер выводит в LST-файл в 30-й колонке символ C (исходный текст в LST-файле начинается с 33-й колонки).

Директива LABEL

Директива LABEL позволяет переопределять атрибут определенного имени.

Формат директивы:

```
имя LABEL тип
```

В качестве типа можно использовать BYTE, WORD или DWORD для переопределения областей данных или имен структур или битовых строк.

Директивой LABEL можно переопределить выполнимый код, как NEAR или FAR.

Эта директива позволяет, например, определить некоторое поле и как DB, и как DW.

Директива NAME

Директива NAME обеспечивает другой способ назначения имени модулю:

```
NAME имя
```

Ассемблер выбирает имя модуля в следующем порядке:

1) если директива NAME присутствует, то ее операнд становится именем модуля;

2) если директива NAME отсутствует, то Ассемблер использует первые шесть символов из директивы TITLE;

3) если обе директивы NAME и TITLE отсутствуют, то именем модуля становится имя исходного файла.

Выбранное имя передается ассемблером в компоновщик.

Директива ORG

Для определения относительной позиции в сегменте данных или кода Ассемблер использует адресный счетчик.

Начальное значение адресного счетчика — 00. Для изменения значения адресного счетчика и соответственно адреса следующего определяемого элемента используется директива ORG. Формат директивы:

```
ORG выражение
```

Выражение может быть абсолютным числом, но не символическим именем, и должно формировать двухбайтовое абсолютное число.

Директива PROC

Любая процедура представляет собой совокупность кодов, начинающуюся директивой PROC и завершающуюся директивой ENDP. Обычно эти директивы используются для подпрограмм в кодовом сегменте. Ассемблер допускает переход на процедуру с помощью команды JMP, но обычной практикой является использование команды CALL для вызова процедуры и RET для выхода из процедуры.

Процедура, находящаяся в одном сегменте с вызывающей процедурой, имеет тип NEAR:

```
имя-процедуры PROC [NEAR]
```

В случае, если операнд опущен, то Ассемблер принимает значение NEAR по умолчанию.

В случае, если процедура является внешней по отношению к вызывающему сегменту, то ее вызов может осуществляться только командой CALL, а сама процедура должна быть объявлена как PUBLIC. Более того, если в вызываемой процедуре используется другое значение ASSUME CS, то необходимо кодировать атрибут FAR:

```
PUBLIC имя-процедуры, имя-процедуры PROC FAR
```

При вызове любой процедуры с помощью команды CALL необходимо обеспечить возврат по команде RET.

Директива PUBLIC

Назначение директивы PUBLIC — информировать ассемблер, что на указанные имена имеются ссылки из других ассемблерных модулей. Формат директивы:

```
PUBLIC имя [,...]
```

Директива RECORD

Директива RECORD позволяет определять битовые строки. Одно из назначений этой директивы — определить однобитовые или многобитовые переключатели. Формат директивы:

```
имя RECORD имя-поля:ширина [=выражение] [, ... ]
```

Имя директивы и имена полей могут быть любыми уникальными идентификаторами. После каждого имени поля следует двоеточие (:) и размер поля в битах, которое может быть от 1 до 16 бит.

Любой размер поля до 8 бит представляется восемью битами, а от 9 до 16 бит — представляется шестнадцатью битами, выровненными справа (если необходимо).

Дополнительно к директиве RECORD имеются операторы WIDTH, MASK и фактор сдвига. Использование этих операторов позволяет изменять определение директивы RECORD без изменения команд, которые имеют ссылки на директиву RECORD.

Оператор WIDTH

Оператор WIDTH возвращает число битов в директиве RECORD или в одном из ее полей.

Фактор сдвига

Прямая ссылка на элемент в RECORD, например:

```
MOV CL,BIT2
```

в действительности не имеет отношения к содержимому BIT2. Вместо этого Ассемблер генерирует непосредственный операнд, который содержит «фактор сдвига», помогающий изолировать необходимое поле. Непосредственное значение представляет собой число, на которое необходимо сдвинуть BIT2 для выравнивания справа.

Оператор MASK

Оператор MASK возвращает «маску» из единичных битовых значений, которые представляют специфицированное поле, иными словами, определяют битовые позиции, которые занимает поле.

Выравнивание

Операнд выравнивания определяет начальную границу сегмента, например

```
PAGE = xxx00
PARA = xxxx0 (граница по умолчанию)
WORD = xxxxe (четная граница)
BYTE = xxxxx
```

где x — любая шестнадцатеричная цифра, е — четная шестнадцатеричная цифра.

Объединение

Операнд объединения указывает способ обработки сегмента, при компоновке:

- NONE: Значение по умолчанию. Сегмент должен быть логически отделен от других сегментов, хотя физически он может быть смежным. Предполагается, что сегмент имеет собственный базовый адрес;
- PUBLIC: Все PUBLIC-сегменты, имеющие одинаковое имя и класс, загружаются компоновщиком в смежные области. Все такие сегменты имеют один общий базовый адрес;
- STACK: Для компоновщика операнд STACK аналогичен операнду PUBLIC. В любой компонуемой программе должен быть определен по крайней мере один сегмент STACK. В случае, если объявлено более одного стека, то стековый указатель (SP) устанавливается на начало первого стека;

- COMMON: Для сегментов COMMON с одинаковыми именами и классами компоновщик устанавливает один общий базовый адрес. При выполнении происходит наложение второго сегмента на первый. Размер общей области определяется самым длинным сегментом;
- AT-параграф: Параграф должен быть определен предварительно. Данный операнд обеспечивает определение меток и переменных по фиксированным адресам в фиксированных областях памяти, таких, как ROM или таблица векторов прерываний в младших адресах памяти.

Класс

Операнд класс может содержать любое правильное имя, заключенное в одиночные кавычки. Данный операнд используется компоновщиком для обработки сегментов, имеющих одинаковые имена и классы.

Типичными примерами являются классы 'STACK' и 'CODE'.

Директива STRUC

Директива STRUC обеспечивает определение различных полей в виде структуры. Данная директива не поддерживается в малом ассемблере ASM.

Формат директивы:

```
Имя-структуры STRUC ...
[определение полей данных] ...
Имя-структуры ENDS
```

Структура начинается собственным именем в директиве STRUC и завершается таким же именем в директиве ENDS.

Ассемблер записывает поля; определенные в структуре, одно за другим от начала структуры.

Правильными операторами определения полей являются DB, DW, DD и DT с указанием имен или без них.



Справочник по командам языка Ассемблера

Обозначение регистров

Команды, использующие регистр, могут содержать три бита, указывающих на конкретный регистр, и один бит «w», определяющий размер регистра: байт или слово. Кроме того, лишь некоторые команды обеспечивают доступ к сегментным регистрам.

Байт способа адресации

Байт способа адресации, если он присутствует, занимает второй байт машинного кода и состоит из следующих трех элементов:

1) mod — двухбитового кода, имеющего значения 11 для ссылки на регистр и 00, 01 и 10 для ссылки на память;

2) reg — трехбитового указателя регистра;

3) r/m — трехбитового указателя регистра или памяти (r — регистр, m — адрес памяти).

Кроме того, первый байт машинного кода может содержать бит «а», который указывает направление потока между операндом 1 и операндом 2.

Биты MOD

Два бита mod определяют адресацию регистра или памяти.

Биты REG

Три бита reg (вместе с битом w) определяют конкретный восьми- или шестнадцатибитовый регистр.

Биты R/M

Три бита r/m (регистр/память) совместно с битами mod определяют способ адресации.

Команды в алфавитном порядке

addr

адрес памяти

addr-high

первый байт адреса (старший)

addr-low

левый (младший) байт адреса

data

непосредственный операнд (8 бит при w=0 и 16 бит при w= 1)

data-high

правый (старший) байт непосредственного операнда

data-low

левый (младший) байт непосредственного операнда

disp

смещение (относительный адрес)

reg

ссылка на регистр.

AAA

Коррекция ASCII-формата для сложения

Операция: Корректирует сумму двух ASCII-байтов в регистре AL. В случае, если правые четыре бита регистра AL имеют значение больше 9 или флаг AF установлен в 1, то команда AAA прибавляет к регистру AH единицу и устанавливает флаги AF и CF. Команда всегда очищает четыре левых бита в регистре AL.

Флаги: Команда воздействует на флаги AF и CF (флаги OF, PF, SF и ZF не определены).

Объектный код: 00110111 (без операндов).

AAD

Коррекция ASCII-формата для деления

Операция: Корректирует ASCII-величины для деления. Команда AAD используется перед делением неупакованных десятичных чисел в регистре AX (удаляет тройки ASCII-кода). Эта команда корректирует де-

лимое в двоичное значение в регистре AL для последующего двоичного деления. Затем умножает содержимое регистра AH на 10, прибавляет результат к содержимому регистра AL и очищает AH. Команда AAD не имеет операндов.

Флаги: Команда воздействует на флаги PF, CF, ZF (флаги AF CF и OF не определены).

Объектный код: |11010101|00001010|.

AAM

Коррекция ASCII-формата для умножения

Операция: Команда AAM используется для коррекции результата умножения двух неупакованных десятичных чисел. Команда делит содержимое регистра AL на 10, записывает частное в регистр AH, а остаток в регистр AL.

Флаги: Команда воздействует на флаги PF, SF и ZF (флаги AF CF и OF не определены).

Объектный код: |11010100|00001010| (без операндов).

AAS

Коррекция ASCII-формата для вычитания

Операция: Корректирует разность двух ASCII-байтов в регистре AL. В случае, если первые четыре бита имеют значение больше 9 или флаг CF установлен в 1, то команда AAS вычитает 6 из регистра AL и 1 из регистра AH, флаги AF и CF при этом устанавливаются в 1. Команда всегда очищает левые четыре бита в регистре AL.

Флаги: Команда воздействует на флаги AF и CF (флаги OF PF SF и ZF не определены).

Объектный код: 00111111 (без операндов).

ADC

Сложение с переносом

Операция: Обычно используется при сложении многословных величин для учета бита переполнения в последующих фазах операции. В случае, если флаг CF установлен в 1, то команда ADC сначала прибавляет 1 к операнду 1. Команда всегда прибавляет операнд 2 к операнду 1, аналогично команде ADD.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

ADD

Сложение двоичных чисел

Операция: Прибавляет один байт или одно слово в памяти, регистре или непосредственно к содержимому регистра или прибавляет один байт или слово в регистре или непосредственно к памяти.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

AND

Логическое И

Операция: Команда выполняет поразрядную конъюнкцию (И) битов двух операндов. Операнды представляют собой одно- или двухбайтовые величины в регистре или памяти. Второй операнд может содержать непосредственные данные. Команда AND проверяет два операнда поразрядно. В случае, если два проверяемых бита равны 1, то в первом операнде устанавливается единичное значение бита, в других случаях — нулевое.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF не определен).

CALL

Вызов процедуры

Операция: Выполняет короткий или длинный вызов процедуры для связи подпрограмм. Для возврата из процедуры используется команда RET. Команда CALL уменьшает содержимое SP на 2 и заносит в стек адрес следующей команды (из IP), а затем устанавливает в регистре IP относительный адрес процедуры. Впоследствии команда RET использует значение в стеке для возврата. Существует четыре типа команды CALL для вызова внутри сегмента и между сегментами. Команда межсегментного вызова сначала уменьшает SP, заносит в стек адрес из регистра CS, а затем загружает в стек внутрисегментный указатель.

Флаги: Не меняются.

CBW

Преобразование байта в слово

Операция: Расширяет однобайтовое арифметическое значение в регистре AL до размеров слова. Команда CBW размножает знаковый бит (7) в регистре AL по всем Битам регистра AH.

Флаги: Не меняются.

Объектный код: 10011000 (без операндов).

CLC

Сброс флага переноса

Операция: Устанавливает значение флага переноса, равное 0, так что, например, команда ADC не прибавляет единичный бит. Команда CLC не имеет операндов.

Флаги: Команда воздействует на флаг CF (устанавливается в 0).

Объектный код: 11111000.

CLD

Сброс флага направления

Операция: Устанавливает значение флага направления, равное 0. В результате такие строковые операции, как CMPS или MOVS обрабатывают данные слева направо.

Флаги: Команда воздействует на флаг DF (устанавливается в 0).

Объектный код: 11111100 (без операндов).

CLI

Сброс флага прерывания

Операция: Запрещает маскируемые внешние прерывания по процессорной шине INTR посредством установки значения флага прерывания IF в 0.

Флаги: Команда воздействует на флаг IF (устанавливается в 0).

Объектный код: 11111010 (без операндов).

CMC

Переключение флага переноса

Операция: Инвертирует флаг CF, то есть, преобразует нулевое значение флага CF в единичное и наоборот.

Флаги: Команда воздействует на флаг CF (инвертируется).

Объектный код: 11110101 (без операндов).

CMP

Сравнение

Операция: Сравнивает содержимое двух полей данных. Фактически команда CMP вычитает второй операнд из первого, но содержимое полей не изменяет. Операнды должны иметь одинаковую длину: байт или слово. Команда CMP может сравнивать содержимое регистра, памяти или непосредственное значение с содержимым регистра; или содержимое регистра или непосредственное значение с содержимым памяти.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

CMPS/CMPSB/CMPSW

Сравнение строк

Операция: Сравнивают строки любой длины. Этим командам обычно предшествует префикс REPn, например REPE CMPSB. Команда CMPSB сравнивает память по байтам, а команда CMPSW — по словам. Первый операнд этих команд адресуется регистровой парой DS:SI, а второй — регистровой парой ES:DI. В случае, если флаг DF установлен в 0, то сравнение происходит слева направо, регистры SI и DI при этом увеличиваются после каждого сравнения. В случае, если флаг DF установлен в 1, то сравнение происходит справа налево, а регистры SI и DI при этом уменьшаются после каждого сравнения.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

Объектный код: 1010011w.

CWD

Преобразование слова в двойное слово

Операция: Расширяет арифметическое значение в регистре AX до размеров двойного слова в регистровой паре DX:AX, дублируя при этом знаковый бит (15-й бит в регистре AX) через регистр DX. Обычно используется для получения 32-битового делимого.

Флаги: Не меняются.

Объектный код: 10011001 (без операндов).

DAA

Десятичная коррекция для сложения

Операция: Корректирует результат сложения двух BCD (десятичных упакованных) элементов в регистре AL. В случае, если четыре правых бита имеют значение больше 9 или флаг AF установлен в 1, то команда DAA прибавляет 6 к регистру AL и устанавливает флаг AF. В случае, если регистр AL содержит значение больше, чем 9F, или флаг CF установлен в 1, то команда DAA прибавляет 60H к регистру AL и устанавливает флаг CF.

Флаги: Команда воздействует на флаги AF. CF, PF, SF и 2F (флаг OF неопределен).

Объектный код: 00100111 (без операндов).

DAS

Десятичная коррекция для вычитания

Операция: Корректирует результат вычитания двух BCD (десятичных упакованных) чисел в регистре AL. В случае, если четыре правых бита имеют значение больше 9 или флаг AF установлен в 1, то команда DAS вычитает 60H из регистра AL и устанавливает флаг CF.

Флаги: Команда воздействует на флаги AF, CF, PF, SF и ZF.

Объектный код: 00101111 (без операндов).

DEC

Декремент

Операция: Вычитает 1 из байта или слова в регистре или в памяти например DEC CX.

Флаги: Команда воздействует на флаги AF, OF, PF, SF и ZF.

DIV

Деление

Операция: Выполняет деление беззнакового делимого (16 или 32 бит) на беззнаковый делитель (8 или 16 бит). Левый единичный бит рассматривается как бит данных, а не как минус для отрицательных чисел. Для 16-битового деления делимое должно находиться в регистре AX, а 8-битовый делитель возможен в регистре или в памяти, например DIV BH. Частное от деления получается в регистре AL, а остаток — в регистре AH. Для 32-битового деления делимое должно находиться в регистровой паре DX:AX а 16-битовый делитель возможен в регистре или в памяти, например DIV CX. Частное от деления получается в регистре AX, а остаток — в регистре DX.

Флаги: Команда воздействует на флаги AF, CF, OF, PF SF и ZF (все не определены).

ESC

Переключение на сопроцессор

Операция: Обеспечивает использование сопроцессора для выполнения специальных операций. Команда ESC передает в сопроцессор инструкцию и операнд для выполнения необходимой операции.

Флаги: Не меняются.

HLT

Останов микропроцессора

Операция: Приводит процессор в состояние останова, в котором происходит ожидание прерывания. При завершении команды HLT регистры CS:IP указывают на следующую команду. При возникновении прерывания процессор записывает в стек регистры CS и IP и выполняет подпрограмму обработки прерывания. При возврате из подпрограммы команда IRET восстанавливает регистры CS и IP из стека и управление передается на команду, следующую за командой HLT.

Флаги: Не меняются.

Объектный код: 11110100 (без операндов).

IDIV

Целое деление знаковых величин

Операция: Выполняет деление знакового делимого (16 или 32 бит) на знаковый делитель (8 или 16 бит). Левый единичный бит рассматривается как знак минус для отрицательных чисел. Для 16-битового деления делимое должно находиться в регистре AX, а 8-битовый делитель возможен в регистре или в памяти, например IDIV DL. Частное от деления получается в регистре AL, а остаток — в регистре AH. Для 32-битового деления делимое должно находиться в регистровой паре DX:AX, а 16-битовый делитель возможен в регистре или в памяти, например IDIV BX. Частное от деления получается в регистре AX, а остаток — в регистре DX.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

IMUL

Целое умножение знаковых величин

Операция: Выполняет умножение на знаковый множитель (8 или 16 бит). Левый единичный бит рассматривается как знак минус для отрицательных чисел. Для 8-битового умножения множимое должно находиться в регистре AL, а множитель возможен в регистре или в памяти, например IMUL BL. Произведение получается в регистре AX. Для 16-битового умножения множимое должно находиться в регистре AX, а множитель возможен в регистре или в памяти, например IMUL BX. Произведение получается в регистровой паре DX:AX.

Флаги: Команда воздействует на флаги CF и OF (флаги AF PF SF и ZF не определены).

IN

Ввод байта или слова из порта

Операция: Передает из вводного порта один байт в регистр AL или два байта в регистр AX). Порт кодируется как фиксированный числовой операнд (IN AX,порт#) или как переменная в регистре DX (IN AX,DX).

Флаги: Не меняются.

INC

Инкремент

Операция: Прибавляет 1 к байту или слову в регистре или в памяти, например INC CX.

Флаги: Команда воздействует на флаги AF, OF, PF, SF и ZF.

INT

Прерывание

Операция: Прерывает выполнение программы и передает управление по одному из 256 адресов (векторов прерывания). Команда INT выполняет следующее:

1) уменьшает значение SP на 2 и заносит в стек флаговый регистр, сбрасывает флаги IF и TF;

2) уменьшает значение SP на 2 и заносит регистр CS в стек, старшее слово из вектора прерывания помещает в регистр CS;

3) уменьшает значение SP на 2 и заносит регистр IP в стек, младшее слово из вектора прерывания помещает в регистр IP.

Флаги: Команда воздействует на флаги IF и TF.

INTO

Прерывание по переполнению

Операция: Приводит к прерыванию при возникновении переполнения (флаг OF установлен в 1) и выполняет команду IRET 4. Адрес подпрограммы обработки прерывания (вектор прерывания) находится по адресу 10H.

Флаги: Не меняются.

Объектный код: 11001110 (без операндов).

IRET

Возврат из обработки прерывания

Операция: Обеспечивает возврат из подпрограммы обработки прерывания. Команда IRET выполняет следующее:

1) помещает слово из вершины стека в регистр IP и увеличивает значение SP на 2;

2) помещает слово из вершины стека в регистр CS и увеличивает значение SP на 2;

3) помещает слово из вершины стека во флаговый регистр и увеличивает значение SP на 2.

Флаги: Команда воздействует на все флаги.

Объектный код: 11001111 (бег операндов).

JA/JNBE

Переход по «выше» или «не ниже или равно»

Операция: Используется после проверки беззнаковых данных для передачи управления по другому адресу. В случае, если флаг CF равен нулю (нет переноса) и флаг ZF равен нулю (не нуль), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JAE/JNB

Переход по «выше или равно» или «не ниже»

Операция: Используется после проверки беззнаковых данных для передачи управления по другому адресу. В случае, если флаг CF равен нулю (нет переноса), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JB/JNAE

Переход по «ниже» или «не выше или равно»

Операция: Используется после проверки беззнаковых данных для Передачи управления по другому адресу. В случае, если флаг CF равен единице (есть перенос), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JBE/JNA

Переход по «ниже или равно» или «не выше»

Операция: Используется после проверки беззнаковых данных для передачи управления по другому адресу. В случае, если флаг CF равен единице (есть перенос) или флаг AF равен единице, то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JC

Переход по переносу

Операция: Идентична JB/JNAE.

JCXZ

Переход по «CX равно нулю»

Операция: Выполняет передачу управления по указанному в операнде адресу, если значение в регистре CX равно нулю. Команда JCXZ может быть полезна в начале циклов LOOP.

Флаги: Не меняются.

JE/JZ

Переход по «равно» или по «нулю»

Операция: Используется после проверки знаковых или беззнаковых данных для передачи управления по другому адресу. В случае, если флаг ZF равен единице (нулевое состояние), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JG/JNLE

Переход по «больше» или «не меньше или равно»

Операция: Используется после проверки знаковых данных для передачи управления по другому адресу. В случае, если флаг ZF равен нулю (не нуль) и флаги SF и OF одинаковы (оба 0 или оба 1), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JGE/JNL

Переход по «больше или равно» или «не меньше»

Операция: Используется после проверки знаковых данных для передачи управления по другому адресу. В случае, если флаги SF и OF одинаковы (оба 0 или оба 1), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JL/JNGE

Переход по «меньше» или «не больше или равно»

Операция: Используется после проверки знаковых данных для передачи управления по другому адресу. В случае, если флаги SF и OF различны, то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JLE/JNG

Переход по «меньше или равно» или «не больше»

Операция: Используется после проверки знаковых данных для передачи управления по другому адресу. В случае, если флаг ZF равен единице (нулевое состояние) и флаги SF и OF различны, то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JMP

Безусловный переход

Операция: Выполняет переход по указанному адресу при любых условиях. Команда JMP заносит в регистр IP необходимый адрес перехода. Существует пять типов команды JMP для передачи управления внутри сегмента или между сегментами. При межсегментном переходе в регистр CS заносится также новый сегментный адрес.

Флаги: Не меняются.

JNC

Переход если нет переноса

Операция: Идентична JAE/JNB.

JNE/JNZ

Переход по «не равно» или по «не нуль»

Операция: Используется после проверки знаковых данных для передачи управления по другому адресу. В случае, если флаг ZF равен нулю (ненулевое состояние), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

INO

Переход, если нет переполнения

Операция: Используется для передачи управления по определенному адресу после проверки на отсутствие переполнения. В случае, если флаг OF равен, нулю (нет переполнения), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JNP/JPO

Переход, если нет паритета или паритет нечетный

Операция: Приводит к передаче управления по определенному адресу, если в результате операции обнаружено отсутствие паритета или паритет нечетный. Нечетный паритет в данном случае означает, что в результате операции в младших восьми битах получено нечетное число битов. В случае, если флаг PF равен нулю (нечетный паритет), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JHS

Переход, если нет знака

Операция: Приводит к передаче управления по определенному адресу, если в результате операции получен положительный знак. В случае, если флаг SF равен нулю (положительное), то команда JNS прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JO

Переход по переполнению

Операция: Приводит к передаче управления по определенному адресу, если в результате операции получено состояние переполнения. В случае, если флаг OF равен единице (переполнение), то команда JO прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JP/JPE

Переход, если есть паритет или паритет четный

Операция: Приводит к передаче управления по определенному адресу, если в результате операции обнаружен четный паритет. Четный паритет в данном случае означает, что в результате операции в младших восьми битах получено четное число битов. В случае, если флаг PF равен единице (четный паритет), то команда прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

JS

Переход по знаку

Операция: Передает управления по определенному адресу, если в результате операции получен отрицательный знак. В случае, если флаг SF равен единице (отрицательно), то команда JS прибавляет к регистру IP значение операнда (относительное смещение) и выполняет таким образом переход.

Флаги: Не меняются.

LAHF

Загрузка флагов в регистр AH

Операция: Загружает значение флагового регистра в регистр AH. Команда LAHF заносит правый байт флагового регистра в регистр AH в следующем виде:

```
SZ * A * P * C (* обозначает неиспользуемые биты)
```

Флаги: Не меняются.

Объектный код: 10011111 (без операндов)

LDS

Загрузка регистра сегмента данных

Операция: Инициализирует начальный адрес сегмента данных и адрес смещения к переменной для обеспечения доступа к данной переменной. Команда LDS загружает в необходимые регистры четыре байта из области памяти, содержащей относительный адрес и сегментный адрес. Сегментный адрес помещается в регистр DS, а относительный адрес — в любой из общих или индексных регистров или в регистровый указатель. Следующая команда загружает относительный адрес в регистр DI:

```
LDS DI,адрес_памяти
```

Флаги: Не меняются.

LES

Загрузка регистра дополнительного сегмента

Операция: Инициализирует начальный адрес дополнительного сегмента и адрес смещения к переменной для обеспечения доступа к данной переменной.

Флаги: Не меняются.

LOCK

Блокировка шины доступа к данным

Операция: Запрещает другим (сопроцессорам одновременно изменять элементы данных. Команда LOCK представляет собой однобайтовый префикс, который можно кодировать непосредственно перед любой командой. Данная операция посылает сигнал в другой процессор, запрещая использование данных, пока не будет завершена следующая команда.

Флаги: Не меняются.

Объектный код: 11110000

LODS/LODSB/LODSW

Загрузка однобайтовой или двухбайтовой строки

Операция: Загружает из памяти один байт в регистр AL или одно слово в регистр AX. Несмотря на то, что команда LODS выполняет строковую операцию, нет смысла использовать ее с префиксом REP. Регистровая пара DS:SI адресует в памяти байт (для LODSB) или слово (для LODSW), которые загружаются в регистр AL или AX соответственно. В случае, если флаг DF равен нулю, то операция прибавляет 1 (для байта) или 2 (для слова) к регистру SI.

В случае, если флаг DF равен единице, то операция вычитает 1 (для байта) или 2 (для слова) из регистра SI.

Флаги: Не меняются.

Объектный код: 1010110w (без операндов).

LOOP

Цикл

Операция: Управляет выполнением группы команд определенное число раз. До начала цикла в регистр CX должно быть загружено число выполняемых циклов. Команда LOOP находится в конце цикла, где она уменьшает значение в регистре CX на единицу. В случае, если значение в регистре CX не равно нулю, то команда передает управление по адресу, указанному в операнде (прибавляет к регистру IP значение операнда); в противном случае управление передается на следующую после LOOP команду (происходит выход из цикла).

Флаги: Не меняются.

LOOPE/LOOPZ

Цикл, если равно или нуль

Операция: Управляет выполнением группы команд определенное число раз или пока установлен флаг ZF (в единичное состояние). Команды LOOPE/LOOPZ аналогичны команде LOOP, за исключением того, что по этим командам цикл прекращается либо по нулевому значению в регистре CX, либо по нулевому значению флага ZF (ненулевое состояние).

Флаги: Не меняются.

LOOPNE/LOOPNZ

Цикл, если не равно или не нуль

Операция: Управляет выполнением группы команд определенное число раз или пока сброшен флаг ZF (в нулевое состояние). Команды LOOPNE/LOOPNZ аналогичны команде LOOP за исключением того, что по этим командам цикл прекращается либо по нулевому значению в регистре CX, либо по единичному значению флага ZF (нулевое состояние).

Флаги: Не меняются.

MOV

Пересылка данных

Операция: Пересылает один байт или одно слово между регистрами или между регистром и памятью, а также передает непосредственное значение в регистр или в память. Команда MOV не может передавать данные между двумя адресами памяти (для этой цели служит команда MOVS). Существует семь типов команды MOV.

Флаги: Не меняются.

MOVS/MOVSB/MOVSW

Пересылка строки байт или строки слов

Операция: Пересылает данные между областями памяти. Команды MOVS(B/W) обычно используются с префиксом REP. Команда MOVSB пересылает любое число байтов, а команда MOVSW — любое число слов. Перед выполнением команды регистровая пара DS:SI должна адресовать источник пересылки («откуда») а регистровая пара ES:DI — получатель пересылки («куда»). В случае, если флаг DF равен нулю, то операция пересылает данные слева направо и увеличивает регистры SI и DI. В случае, если флаг DF равен единице то операция пересылает данные справа налево и уменьшает регистры SI и DI.

Флаги: Не меняются.

Объектный код: 1010010w (без операндов).

MUL

Беззнаковое умножение

Операция: Умножает беззнаковое множимое (8 или 16 бит) на беззнаковый множитель (8 или 16 бит). Левый единичный бит рассматривается как бит данных, но не как знак минус для отрицательных чисел. Для 8-битового умножения множимое должно находиться в регистре AL, а множитель возможен в регистре или в памяти, например MUL CL. Произведение получается в регистре AX. Для 16-битового умножения множимое должно находиться в регистре AX, а множитель возможен в регистре или в памяти, например MUL BX. Произведение получается в регистровой паре DX:AX.

Флаги: Команда воздействует на флаги CF и OF (флаги AF, PF, SF и ZF не определены).

NEG

Изменение знака числа

Операция: Меняет двоичное значение из положительного в отрицательное и из отрицательного в положительное. Команда NEG вычисляет двоичное дополнение от указанного операнда: вычитает операнд из нуля и прибавляет единицу. Операндом может быть байт или слово в регистре или в памяти.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

NOP

Нет операции

Операция: Применяется для удаления или вставки машинных кодов или для задержки выполнения программы. Команда NOP выполняет операцию XCHG AX,AX, которая ничего не меняет.

Флаги: Не меняются.

Объектный код: 10010000 (без операндов)

NOT

Логическое НЕТ

Операция: Меняет нулевые биты на единичные и наоборот. Операндом может быть байт или слово в регистре или в памяти.

Флаги: Не меняются.

OR

Логическое ИЛИ

Операция: Выполняет поразрядную дизъюнкцию (ИЛИ) над битами двух операндов. Операндами являются байты или слова в регистрах или в памяти, второй операнд может иметь непосредственное значение. Команда OR обрабатывает операнды побитово. В случае, если любой из проверяемых бит равен единице, то бит в операнде 1 становится равным единице, в противном случае бит в операнде 1 не изменяется.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF неопределен).

OUT

Вывод байта или слова в порт

Операция: Передает в выводной порт байт из регистра AL или слово из регистра AX. Порт кодируется как фиксированный числовой опе-

ранд (OUT порт#,AX) или как переменная в регистре DX (OUT DX.AX). Процессоры имеют, кроме того, команду OUTS (Output String — вывод строки).

Флаги: Не меняются.

POP

Извлечение слова из стека

Операция: Передает слово (помещенное ранее в стек) в указанный операнд. Регистр SP указывает на текущее слово в вершине стека. Команда POP извлекает слово из стека и увеличивает значение в регистре SP на 2. Существует три типа команды POP в зависимости от операнда: общий регистр, сегментный регистр, слово в памяти.

Флаги: Не меняются.

POPA

Извлечение из стека всех общих регистров

Операция: Извлекает из стека восемь значений в регистры Dl, SI BP, SP, BX, DX, CX, AX в указанной последовательности и увеличивает регистр SP на 16. Регистры обычно записываются в стек ранее соответствующей командой PUSHA.

Флаги: Не меняются.

Объектный код: 01100001 (без операндов).

POPF

Извлечение флагов из стека

Операция: Передает биты (помещенные ранее в стек) во флаговый регистр. Регистр SP указывает на текущее слово в вершине стека. Команда POPF передает биты из этого слова во флаговый регистр и увеличивает значение в регистре SP на 2. Обычно команда PUSHF записывает значения флагов в стек, а команда POPF восстанавливает эти флаги.

Флаги: Команда воздействует на все флаги.

Объектный код: 10011101 (без операндов).

PUSH

Занесение слова в стек

Операция: Сохраняет значение слова (адрес или элемент данных) в стеке для последующего использования. Регистр SP указывает на текущее слово в вершине стека. Команда PUSH уменьшает значение в регистре SP на 2 и передает слово из указанного операнда в новую вершину стека. Существует три типа команды PUSH в зависимости от операнда: общий регистр, сегментный регистр или слово в памяти.

Флаги: Не меняются.

PUSHA

Занесение в стек всех общих регистров

Операция: Записывает в стек восемь значений регистров AX, CX, DX, BX, SP, BP, SI, DI в указанной последовательности и уменьшает регистр SP на 16. Обычно команда POPA позже восстановит эти регистры из стека.

Флаги: Не меняются.

Объектный код: 01100000 (без операндов).

PUSHF

Занесение флагов в стек

Операция: Сохраняет значения флагов из флагового регистра в стеке для последующего использования. Регистр SP указывает на текущее слово в вершине стека. Команда PUSHF уменьшает значение в регистре SP на 2 и передает флаги в новую вершину стека.

Флаги: Не меняются.

Объектный код: 10011100 (без операндов).

RCL и RCR

Циклический сдвиг влево через перенос и циклический сдвиг вправо через перенос

Операция: Выполняет циклический сдвиг битов (ротацию) влево или вправо через флаг CF. Данные операции могут выполняться в байте или в слове, в регистре или в памяти. Ротация на один бит кодируется в команде значением 1; ротация более чем на один бит требует указания регистра CL, который содержит счетчик. Для команды RCL значение флага CF записывается в бит 0, а. самый левый бит записывается во флаг CF; все другие биты сдвигаются влево. Для команды RCR значение флага CF записывается в самый левый бит, а бит 0 записывается во флаг CF; все другие биты сдвигаются вправо.

Флаги: Команда воздействует на флаги CF и OF.

REP/REPE/REPZ/REPNE/REPNZ

Повтор строковой операции

Операция: Повторяет строковую операцию определенное число раз. Используется в качестве префикса повторения перед строковыми командами CMPS, MOVS, SCAS, STOS. Счетчик повторений должен быть загружен в регистр CX до выполнения строковой команды. Операция уменьшает регистр CX на 1 при каждом выполнении строковой команды. Для префикса REP операция повторяется, пока содержимое регистра CX не достигнет нуля. Для префикса REPE/REPZ операция повторяется, пока регистр CX содержит ненулевое значение и флаг ZF равен 1 (нулевое состояние). Для префикса REPNE/REPNZ операция повторяется, пока регистр CX содержит ненулевое значение и флаг ZF равен 0 (ненулевое состояние).

Флаги: Определяются соответствующей строковой командой.

Объектный код: REP/REPNE: 11110010 REPE: 11110011

RET

Возврат из процедуры

Операция: Возвращает управление из процедуры, вызванной ранее командой CALL. Команда CALL может передавать управление внутри одного сегмента или между сегментами. Команда RET заносит слово из вершины стека в регистр IP и увеличивает значение SP на 2. Для межсегментного возврата команда RET, кроме того, заносит слово из новой вершины стека в регистр CS и еще раз увеличивает значение SP на 2. Любой числовой операнд команды (например, RET 4) прибавляется к указателю стека SP.

Флаги: Не меняются.

ROL и ROR

Циклический сдвиг влево и циклический сдвиг вправо

Операция: Выполняет циклический сдвиг битов (ротацию) влево или вправо. Данные операции могут выполняться в байте или в слове, в регистре или в памяти. Ротация на один бит кодируется в команде значением 1; ротация более чем на один бит требует указания регистра CL, который содержит счётчик. Для команды ROL самый левый бит записывается в бит 0; все другие биты сдвигаются влево. Для команды ROR бит 0 записывается в самый левый бит; все другие биты сдвигаются вправо.

Флаги: Команда воздействует на флаги CF и OF.

SAHF

Установка флагов из регистра AH

Операция: Данная команда обеспечивает совместимость с процессором 8080 для пересылки значений флагов из регистра AH во флаговый регистр. Команда SAHF пересылает определенные биты из регистра AH во флаговый регистр в следующем виде:

```
SZ * A * P * C (* обозначает неиспользуемые биты)
```

Флаги: Не меняются.

Объектный код: 10011110 (без операндов)

SAL, SAR, SHL и SHR

Сдвиг влево или вправо

Операция: Выполняет сдвиг битов влево или вправо. Данные операции могут выполняться в байте или в слове, в регистре или в памяти. Сдвиг на один бит кодируется в команде значением 1; сдвиг более чем на один бит требует указания регистра CL, который содержит счетчик сдвига.

Команда SAR выполняет арифметический сдвиг, который учитывает знак сдвигаемого значения. Команды SHL и SHR выполняют логический сдвиг и рассматривают знаковый бит как обычный бит данных. Команда SAL выполняется аналогично команде SHL. Команды SAL и SHL сдвигают биты влево определенное число раз и правый освобождающийся бит заполняют нулевым значением.

Команда SHR сдвигает биты вправо определенное число раз и левый освобождающийся бит заполняет нулевым значением. Команда SAR сдвигает биты вправо определенное число раз и левый освобождающийся бит заполняет значением знакового бита (0 или 1). Во всех случаях значения битов, выдвигаемых за разрядную сетку, теряются.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF не определен).

SBB

Вычитание с заемом

Операция: Обычно используется при вычитании многословных двоичных величин для учета единичного бита переполнения в последующей фазе операции. В случае, если флаг CF установлен в 1, то команда SBB сначала вычитает 1 из операнда 1. Команда SBB всегда вычитает операнд 2 из операнда 1, аналогично команде SUB.

Флаги: Команда воздействует на флаги AF, CF, OF, PF. SF и ZF.

SCAS/SCASB/SCASW

Поиск байта или слова в строке

Операция: Выполняет поиск определенного байта или слова в строке. Для команды SCASB необходимое значение загружается в регистр AL, а для команды SCASW — в регистр AX. Регистровая пара ES:DI указывает на строку в памяти, которая должна быть сканирована. Данные команды обычно используются с префиксом REPE или REPNE. В случае, если флаг DF равен нулю, то операция сканирует память слева направо и увеличивает регистр DI. В случае, если флаг DF равен единице, то операция сканирует память справа налево и уменьшает регистр DI.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

Объектный код: 1010111w (без операндов).

STC

Установка флага переноса

Операция: Устанавливает значение флага CF в 1.

Флаги: Команда воздействует на флаг CF (устанавливается в 1).

Объектный код: 11111001 (без операндов).

STD

Установка флага направления

Операция: Устанавливает значение флага направления в 1. В результате строковые операции, такие, как MOVS или CMPS, обрабатывают данные справа налево.

Флаги: Команда воздействует на флаг DF (устанавливается в 1).

Объектный код: 11111101 (без операндов).

STI

Установка флага прерывания

Операция: Разрешает маскируемые внешние прерывания после выполнения следующей команды и устанавливает значения флага прерывания IF в 1.

Флаги: Команда воздействует на флаг IF (устанавливается в 1).

Объектный код: 11111011 (без операндов).

STOS/STOSB/STOSW

Запись однобайтовой или двухбайтовой строки

Операция: Сохраняет байт или слово в памяти. При использовании префикса REP операция дублирует значение байта или слова определенное число раз, что делает ее удобной для очистки областей памяти. Для команды STOSB необходимый байт загружается в регистр AL, а для команды STOSW необходимое слово загружается в регистр AX. Регистровая пара ES:DI указывает область памяти, куда должен быть записан байт или слово. В случае, если флаг DF равен нулю, то операция записывает в память слева направо и увеличивает регистр DI. В случае, если флаг DF равен единице, то операция записывает в память справа налево и уменьшает значение в регистре DI.

Флаги: Не меняются.

Объектный код: 1010101w (без операндов).

SUB

Вычитание двоичных чисел

Операция: Вычитает байт или слово в регистре, памяти или непосредственное значение из регистра; или вычитает байт или слово в регистре или непосредственное значение из памяти.

Флаги: Команда воздействует на флаги AF, CF, OF, PF, SF и ZF.

TEST

Проверка битов

Операция: Команда выполняет проверку байта или слова на определенную битовую комбинацию. Команда TEST действует аналогично команде AND, но не изменяет результирующий операнд. Операнды могут иметь однобайтовые или двухбайтовые значения. Второй операнд может иметь непосредственное значение. Команда выставляет флаги в соответствии с логической функцией И.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF не определен).

WAIT

Установка процессора в состояние ожидания

Операция: Позволяет процессору оставаться в состоянии ожидания, пока не произойдет внешнее прерывание. Данная операция необходима для обеспечения синхронизации процессора с внешним устройством или с сопроцессором. Процессор ожидает, пока внешнее устройство

(или сопроцессор) не закончит выполнение операции и на входной линии TEST не появится сигнал (активный уровень).

Флаги: Не меняются.

Объектный код: 10011011

XCHG

Перестановка

Операция: Переставляет два байта или два слова между двумя регистрами (например, XCHG AH,BL) или между регистром и памятью (например, XCHG CX,word).

Флаги: Не меняются.

XLAT

Перекодировка

Операция: Транслирует байты в другой формат, например при переводе нижнего регистра в верхний или при перекодировке ASCII-кода в EBCDIC-код. Для выполнения данной команды необходимо определить таблицу преобразования байт и загрузить ее адрес в регистр BX. Регистр AL должен содержать байт, который будет преобразован с помощью команды XLAT. Операция использует значение в регистре AL как смещение в таблице, выбирает байт по этому смещению и помещает его в регистр AL.

Флаги: Не меняются.

Объектный код: 11010111 (без операндов).

XOR

Исключающее ИЛИ

Операция: Выполняет логическую операцию исключающего ИЛИ над битами двух операндов. Операндами являются байты или слова в регистрах или в памяти, второй операнд может иметь непосредственное значение. Команда XOR обрабатывает операнды побитово. В случае, если проверяемые биты одинаковы, то команда XOR устанавливает бит в операнде 1 равным нулю, если биты различны, то бит в операнде 1 устанавливается равным единице.

Флаги: Команда воздействует на флаги CF, OF, PF, SF и ZF (флаг AF неопределен).

Список использованной литературы

Язык Ассемблера для IBM PC и программирования

Питер Абель. Перевод с английского Ю.В. Сальникова

Системное программирование

Дж. Джордан

Included documentation and sources

Borland

Разработка общего программного обеспечения

Московский Государственный Институт Электроники и Математики

Научно-популярное издание

Фельдман Сергей Константинович

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ НА ПЕРСОНАЛЬНОМ КОМПЬЮТЕРЕ

Главный редактор *Б. К. Леонтьев*
Шеф-редактор *А. Г. Бенеташвили*
Оригинал-макет *И. В. Царик*
Художник *О. К. Алехин*
Художественный редактор *М. Л. Мишин*
Технический редактор *К. В. Шапиро*
Корректоры *Л. С. Зимина, К. В. Толкачева*

Подписано в печать 07.05.2006. Формат 60×90/16.
Гарнитура «Ньютон». Бумага офсетная. Печать офсетная.
Печ. л. 32. Тираж 3000.